CONCEPTUAL

The network correctness and performance testing language

Scott Pakin, pakin@lanl.gov

This document describes coNCePTuaL version 0.5.3.

Table of Contents

1 Introduction

This document presents a simple, special-purpose language called coNCePTuaL. coNCePTuaL is intended for rapidly generating programs that measure the performance and/or test the correctness of networks and network protocol layers. A few lines of coNCePTuaL code can produce programs that would take significantly more effort to write in a conventional programming language.

coNCePTuaL is not merely a language specification. The coNCePTuaL toolset includes a compiler, run-time library, and associated utility programs that enable users to analyze network behavior quickly, conveniently, and accurately.

1.1 Motivation

A frequently reinvented wheel among network researchers is a suite of programs that test a network's performance. A problem with having umpteen versions of performance tests is that it leads to a variety in the way results are reported; colloquially, apples are often compared to oranges. Consider a bandwidth test. Does a bandwidth test run for a fixed number of iterations or a fixed length of time? Is bandwidth measured as ping-pong bandwidth (i.e., $2 \times$ message length $\div$ round-trip time) or unidirectional throughput (N messages in one direction followed by a single acknowledgement message)? Is the acknowledgement message of minimal length or as long as the entire message? Does its length contribute to the total bandwidth? Is data sent unidirectionally or in both directions at once? How many warmup messages (if any) are sent before the timing loop? Is there a delay after the warmup messages (to give the network a chance to reclaim any scarce resources)? Are receives nonblocking (possibly allowing overlap in the NIC) or blocking?

The motivation behind creating coNCePTuaL, a simple specification language designed for describing network benchmarks, is that it enables a benchmark to be described sufficiently tersely as to fit easily in a report or research paper, facilitating peer review of the experimental setup and timing measurements. Because coNCePTuaL code is simple to write, network tests can be developed and deployed with low turnaround times—useful when the results of one test suggest a following test that should be written. Because coNCePTuaL is special-purpose its run-time system can perform the following functions, which benchmark writers often neglect to implement:

- logging information about the environment under which the benchmark ran: operating system, CPU architecture and clock speed, timer type and resolution, etc.
- aborting a program if it takes longer than a predetermined length of time to complete
- writing measurement data and descriptive statistics to a variety of output formats, including the input formats of various graph-plotting programs

coNCePTuaL is not limited to network peformance tests, however. It can also be used for network verification. That is, coNCePTuaL programs can be used to locate failed links or to determine the frequency of bit errors—even those that may sneak past the network's CRC hardware.

In addition, because coNCePTuaL is a very high-level language, the coNCePTuaL compiler's backend has a great deal of potential. It would be possible for the backend to produce a variety of target formats such as Fortran + MPI, Perl + sockets, C + a network

vendor's low-level messaging layer, and so forth. It could directly manipulate a network simulator. It could feed into a graphics program to produce a space-time diagram of a coNCePTuaL program. The possibilities are endless.

1.2 Typesetting conventions

The following table showcases the typesetting conventions used in this manual to attribute various meanings to text. Note that not all of the conventions are typographically distinct.

`-a` `--abcdef`	command-line options (e.g., `-C` or `--help`)
ABCDEF	environment variables (e.g., *PATH*)
⟨*abcdef*⟩	nonterminals in the coNCePTuaL grammar (e.g., ⟨*ident*⟩)
abcdef	commands to enter on the keyboard (e.g., *make install*)
'abcdef'	file and directory names (e.g., 'conceptual.pdf')
ABCDEF	coNCePTuaL keywords (e.g., RECEIVE)
abcdef	variables, constants, functions, and types in any language (e.g., bit_errors or gettimeofday())
abcdef	metasyntactic variables and formal function parameters (e.g., *fan-out*)
'abcdef'	snippets of code, command lines, files, etc. (e.g., '10 MOD 3')

2 Installation

coNCePTuaL uses the GNU Autotools (Autoconf, Automake, and Libtool) to increase portability, to automate compilation, and to facilitate installation. As of this writing, coNCePTuaL has passed *`make check`* (see Section 2.2 [make], page 5) on the following platforms:

Architecture	OS	Compiler
IA-32	Linux	'gcc' (GNU)
		'icc' (Intel)
		'pgcc' (PGI)
	FreeBSD	'gcc' (GNU)
	OpenBSD	'gcc' (GNU)
	Windows (via Cygwin)	'gcc' (GNU)
IA-64	Linux	'gcc' (GNU)
		'ecc' (Intel)
PowerPC	Linux	'gcc' (GNU)
		'xlc' (IBM)
	MacOS X	'gcc' (GNU)
	BLRTS	'xlc' (IBM)
Cray X1	UNICOS/mp	'cc' (Cray)
UltraSPARC	Solaris	'gcc' (GNU)
		'cc' (Sun)
MIPS	IRIX	'gcc' (GNU)
		'cc' (MIPSpro)
Alpha	Linux	'gcc' (GNU)
		'ccc' (HP)
	Tru64	'gcc' (GNU)
		'cc' (HP)

In its simplest form, coNCePTuaL installation works by executing the following commands at the operating-system prompt:

```
./configure
make
make install
```

('`configure`' is normally run as *`./configure`* to force it to run from the current directory on the assumption that '`.`' is not in the executable search path.) We now describe those three installation steps in detail, listing a variety of customization options for each step.

2.1 *configure*

'`configure`' is a Bourne-shell script that analyzes your system's capabilities (compiler features, library and header-file availability, function and datatype availability, linker flags for various options, etc.) and custom-generates a '`Makefile`' and miscellaneous other files. '`configure`' accepts a variety of command-line options. *`./configure --help`* lists all of the options. The following are some of the more useful ones:

`--disable-shared`
: coNCePTuaL normally installs both static and dynamic libraries. While dynamic libraries have a number of advantages they do need to be installed on all nodes that run the compiled coNCePTuaL programs. If global installation is not convenient/feasible, *`--disable-shared`* can be used to force static linking of executables. Note, however, that '`libncptlmodule.so`', the Python interface to the coNCePTuaL run-time library, needs to be built as a shared object so that it can be loaded dynamically into a running Python interpreter. *`--disable-shared`* inhibits the compilation and installation of '`libncptlmodule.so`'.

`--prefix=`directory
: *`make install`* normally installs coNCePTuaL into the '`/usr/local`' directory. The *`--prefix`* option instructs '`configure`' to write a '`Makefile`' with a different installation directory. For example, *`--prefix=/local/encap/conceptual-0.5.3`* will cause coNCePTuaL's files to be installed in '`/local/encap/conceptual-0.5.3/bin`', '`/local/encap/conceptual-0.5.3/include`', etc.

`--disable-papi`
: coNCePTuaL normally uses the Performance API (PAPI)—if available—to help acquire system information. However, if PAPI cannot be installed on all of the nodes that will be running compiled coNCePTuaL programs then it may be worth instructing coNCePTuaL not to use PAPI, even if available.

`--with-gettimeofday`
: The coNCePTuaL run-time library is able to use any of a variety of platform-specific microsecond timers to take to timing measurements. The *`--with-gettimeofday`* option forces the run-time library to utilize instead the generic C `gettimeofday()` function. This can be useful in the rare, but not impossible, case that a quirk in some particular platform misleads one of coNCePTuaL's other timers. The '`validatetimer`' utility (see Section 2.2 [make], page 5) can help determine whether *`--with-gettimeofday`* is necessary.

`CC=`C compiler
: '`configure`' automatically searches for a C compiler to use. To override its selection, assign a value to *`CC`* on the command line. For example, *`./configure CC=ecc`* will cause coNCePTuaL to be built with '`ecc`'.

CFLAGS=C compiler flags
LDFLAGS=linker flags
CPPFLAGS=C preprocessor flags
LIBS=extra libraries

Like *CC*, these variables override the values determined automatically by '`configure`'. As an illustration, *./configure CPPFLAGS="-DSPECIAL -I/home/pakin/include/special -I." CFLAGS="-O3 -g -Wall -W" LDFLAGS=--static LIBS="-lz /usr/lib/libstuff.a"* assigns values to all four variables.

MPICC=C compiler
MPICPPFLAGS=C preprocessor flags
MPILDFLAGS=extra linker flags
MPILIBS=extra libraries

These variables are analagous to *CC*, *CPPFLAGS*, *LDFLAGS*, and *LIBS*, respectively. The difference is that they are not used to build the coNCePTuaL run-time library but rather to build user programs targeted to the C+MPI compiler backend. For example, if your MPI installation lacks an '`mpicc`' script, you may need to specify extra header files and libraries explicitly: *./configure MPICPPFLAGS="-I/usr/lib/mpi/include" MPILIBS="-lmpich"*.

As a rather complex illustration of how some of the preceding options (as well as a few mentioned by *./configure --help*) might be combined, the following is how coNCePTuaL was once configured to cross-compile from a Linux/PowerPC build machine to a prototype of the BlueGene/L supercomputer (containing, at the time, 2048 embedded PowerPC processors, each executing a minimal run-time system, BLRTS). IBM's '`xlc`' compiler was accessed via a wrapper script called '`mpcc`'.

```
./configure CFLAGS="-g -O -qmaxmem=64000" CC=/bgl/local/bin/mpcc
CPP="gcc -E" --host=powerpc-ibm-linux-gnu --build=powerpc-unknown-linux-gnu
--with-alignment=8 --with-gettimeofday --prefix=/bgl/bgguest/LANL/ncptl
MPICC=/bgl/local/bin/mpcc CPPFLAGS=-I/BlueLight/floor/bglsys/include
```

It's always best to specify environment variables as arguments to *./configure* because the '`configure`' script writes its entire command line as a comment to '`config.log`' and as a shell command to '`config.status`' to enable re-running *./configure* with exactly the same parameters.

When *./configure* finishes running it outputs a list of the warning messages that were issued during the run. If no warnings were issued, *./configure* will output '`Configuration completed without any errors or warnings.`'.

2.2 *make*

Running *make* by itself will compile the coNCePTuaL run-time library. However, the '`Makefile`' generated by '`configure`' can perform a variety of other actions, as well:

make check

Perform a series of regression tests on the coNCePTuaL run-time library. This is a good thing to do after a *make* to ensure that the run-time library

built properly on your system. If any tests fail, it may be possible to gain more information about the source of the problem by compiling the test suite with the `DEBUG` symbol defined in the C preprocessor and re-running it:

```
cd tests
make clean
make CPPFLAGS="-DDEBUG" check
```

make clean
make distclean
make maintainer-clean

make clean deletes all files generated by a preceding *make* command. *make distclean* deletes all files generated by a preceding *./configure* command. *make maintainer-clean* delete all generated files. Run *make maintainer-clean* only if you have fairly recent versions of the GNU Autotools (Autoconf 2.53, Automake 1.6, and Libtool 1.4) because those are needed to regenerate some of the generated files. The sequence of operations to regenerate all of the configuration files needed by coNCePTuaL is shown below.

```
libtoolize --force --copy
aclocal
autoheader
automake --add-missing --copy
autoconf
```

make install

Install coNCePTuaL, including the compiler, run-time library, header files, and tools. *make install* is described in detail later in this section.

make uninstall

Remove all of the files that *make install* installed. Most of the top-level directories are retained, however, as '`make`' cannot guarantee that these are not needed by other applications.

make info
make pdf
make docbook

Produce the coNCePTuaL user's manual (this document) in, respectively, Emacs info format, PDF format, or DocBook format. The resulting documentation ('`conceptual.info*`', '`conceptual.pdf`', or '`conceptual.xml`') is created in the '`doc`' subdirectory.

make logextract.html

coNCePTuaL comes with a postprocessor called '`logextract`' which facilitates extracting information from coNCePTuaL-produced log files. The complete '`logextract`' documentation is presented in Section 3.4.2 [logextract], page 28. As is readily apparent from that documentation, '`logextract`' supports an overwhelming number of command-line options. To make the '`logextract`' documentation more approachable, the *make logextract.html* command creates a dynamic HTML version of it (and stores in the '`doc`'

subdirectory). The result, '`logextract.html`', initially presents only the top level of the '`logextract`' option hierarchy. Users can then click on the name of a command-line option to expand or contract the list of subobtions. This interactive behavior makes it easy for a user to get more information on some options without being distracted by the documentation for the others.

make empty.log

Create an empty log file called '`empty.log`' which contains a complete header and trailer but no data. This is convenient for validating that the coNCePTuaL run-time library was built using your preferred build options.

make stylesheets

coNCePTuaL can automatically produce stylesheets for a variety of programs. These stylesheets make keywords, comments, strings, and other terms in the language visually distinct from each other for a more aesthetically appealing appearance. Currently, *make stylesheets* produces a LaTeX 2$_\varepsilon$ package ('`ncptl.sty`'), an '`a2ps`' style sheet ('`ncptl.ssh`'), an Emacs major mode ('`ncptl-mode.el`'/'`ncptl-mode.elc`'), and a Vim syntax file ('`ncptl.vim`'). Note that the '`Makefile`' currently lacks provisions for installing these files so whichever stylesheets are desired will need to be installed manually. Stylesheet installation is detailed at the end of this section.

make modulefile

The Modules package (`http://modules.sourceforge.net/`) facilitates configuring the operating-system shell for a given application. The *make modulefile* command creates a '`conceptual_0.5.3`' modulefile that checks for conflicts with previously loaded coNCePTuaL modulefiles then sets the *PATH*, *MANPATH*, and *LD_LIBRARY_PATH* environment variables to values appropriate values as determined by '`configure`' (see Section 2.1 [configure], page 4).

Normally, '`conceptual_0.5.3`' should be installed in the system's module path (as described by the *MODULEPATH* environment variable). However, users without administrator access can still use the coNCePTuaL modulefile as a convenient mechanism for properly setting all of the environment variables needed by coNCePTuaL:

```
make modulefile
module load ./conceptual_0.5.3
```

See the '`module`' man page for more information about modules.

make dist

Package together all of the files needed to rebuild coNCePTuaL. The resulting file is called '`conceptual-0.5.3.tar.gz`' (for this version of coNCePTuaL).

make all

Although `all` is the default target it can also be specified explicitly. Doing so is convenient when performing multiple actions at once, e.g., *make clean all*.

make tags

Produce/update a '`TAGS`' file that the Emacs text editor can use to find function declarations, macro definitions, variable definitions, `typedef`s, etc. in the coNCePTuaL run-time library source code. This is useful primarily for developers wishing to interface with the coNCePTuaL run-time library. Read the Emacs documentation for *`M-x find-tag`* for more information.

Validating the coNCePTuaL timer

`make` automatically builds a program called '`validatetimer`'. '`validatetimer`' helps validate that the real-time clock used by the coNCePTuaL run-time library accurately measures wall-clock time. The idea is to compare coNCePTuaL's timer to an external clock (i.e., one not associated with the computer). Simply follow the program's prompts:

```
% validatetimer
Press <Enter> to start the clock ...
Press <Enter> again in exactly 60 seconds ...

coNCePTuaL measured 60.005103 seconds.
coNCePTuaL timer error = 0.008505%
```

If the difference between coNCePTuaL's timer and an external clock is significant, then performance results from coNCePTuaL—and possibly from other programs, as well—should not be trusted. Note that only extreme differences in timings are significant; there will always be *some* error caused by human response time and by system I/O speed. In the case that there *is* an extreme performance difference,[1] the *`--with-gettimeofday`* option to '`configure`' (see Section 2.1 [configure], page 4) may be a viable workaround.

'`validatetimer`' takes an optional command-line argument, which is the number of seconds of wall-clock time to expect. The default is '`60`'. Larger numbers help amortize error; smaller numbers enable the program to finish sooner.

2.3 *`make install`*

The coNCePTuaL compiler and run-time library are installed with *`make install`*. Although '`configure`' can specify the default installation directory this can be overridden at *`make install`* time in one of two ways. *`make DESTDIR=`*`prefix` *`install`* prepends *prefix* to every directory when installing. However, the files are installed believing that *`DESTDIR`* was not specified. For example, *`make DESTDIR=/mnt install`* would cause executables to be installed into '`/mnt/usr/local/bin`', but if any of these are symbolic links, the link will omit the '`/mnt`' prefix.

The second technique for overriding installation directories is to specify a new value for '`prefix`' on the command line. That is, *`make prefix=/opt/ncptl install`* will install

[1] To date, extreme performance differences have been observed primarily on PowerPC-based systems. The PowerPC cycle counter is clocked at a different rate from the CPU speed, which may confuse coNCePTuaL. The run-time library compensates for this behavior on all tested platforms (see Chapter 2 [Installation], page 3), but the user should nevertheless make sure to run '`validatetimer`' to verify that coNCePTuaL's timer is sufficiently accurate.

into '/opt/ncptl/bin', '/opt/ncptl/include', '/opt/ncptl/man', etc., regardless of the *--prefix* value given to 'configure'. coNCePTuaL's 'Makefile' provides even finer-grained control than that. Instead of—or in addition to—specifying a *prefix* option on the command line, individual installation directories can be named explicitly. These include *bindir*, *datadir*, *libdir*, *includedir*, *infodir*, *mandir*, *pkgdatadir*, *pythondir*, and many others. Scrutinize the 'Makefile' to find a particular directory that should be overridden.

The remainder of this section presents a number of optional installation steps that add coNCePTuaL support to a variety of third-party software packages.

Installing stylesheets

The *make stylesheets* command produces a variety of stylesheets for presenting coNCePTuaL code in a more pleasant format than ordinary, monochromatic text. Stylesheets must currently be installed manually as per the following instructions:

'ncptl.sty'
: 'ncptl.sty' is typically installed in '*texmf*/tex/latex/misc', where *texmf* is likely to be '/usr/local/share/texmf'. On a Web2c version of TEX the command *kpsewhich -expand-var='$TEXMFLOCAL'* should output the correct value of *texmf*. In most TEX distributions the filename database needs to be refreshed after a new package is installed. See http://www.tex.ac.uk/cgi-bin/texfaq2html?label=instpackages for more information. 'ncptl.sty' is merely a customization of the 'listings' package that defines a new language called 'ncptl'. See the 'listings' documentation for instructions on typesetting source code.

'ncptl.ssh'
: Running *a2ps --list=defaults* outputs (among other things) the 'a2ps' library path. 'ncptl.ssh' should be installed in one of the 'sheets' directories listed there, typically '/usr/share/a2ps/sheets'.

'ncptl-mode.el'
'ncptl-mode.elc'
: 'ncptl-mode.el' and 'ncptl-mode.elc' belong in a local Elisp directory that is part of the Emacs load-path, e.g., '/usr/share/emacs/site-lisp'. The following Elisp code, which belongs in '~/.emacs' for GNU Emacs or '~/.xemacs/init.el' for XEmacs, makes Emacs set ncptl-mode whenever opening a file with extension '.ncptl':

```
(autoload 'ncptl-mode "ncptl-mode"
  "Major mode for editing coNCePTuaL programs." t)
(add-to-list 'auto-mode-alist '("\\.ncptl$" . ncptl-mode))
```

: Syntax highlighting should be enabled by default. If it isn't, the Emacs command *M-x font-lock-mode* should enable it for the current buffer.

‘`ncptl.vim`’

Vim's syntax-file directory may be named after the Vim version, e.g., ‘`/usr/share/vim/vim61/syntax`’ for Vim 6.1. Put ‘`ncptl.vim`’ there. To associate ‘`.ncptl`’ files with coNCePTuaL code, the following lines need to be added to Vim's ‘`filetype.vim`’ file somewhere between the ‘`augroup filetypedetect`’ line and the ‘`augroup END`’ line:

```
" coNCePTuaL
au BufNewFile,BufRead *.ncptl           setf ncptl
```

SLOCCount

SLOCCount (`http://www.dwheeler.com/sloccount/`) is a utility that counts the number of lines of code in a file, excluding blank lines and comments. SLOCCount supports a variety of programming languages and it is straightforward to get it to support coNCePTuaL, as well. The procedure follows the “Adding support for new languages” section of the SLOCCount manual:

1. Create an ‘`ncptl_count`’ script with the following contents:

```
#! /bin/sh

generic_count "#" $@
```

2. Mark the script executable and install it somewhere in your executable search path.
3. Edit SLOCCount's ‘`break_filelist`’ Perl script to include the following association in the `%file_extensions` hash:

```
  "ncptl" => "ncptl",    # coNCePTuaL
```

pkg-config

The ‘`pkg-config`’ utility helps ensure that programs are given appropriate compiler and linker flags to use a particular package's C header files and libraries. coNCePTuaL's ‘`configure`’ script (see Section 2.1 [configure], page 4) automatically produces a ‘`pkg-config`’ configuration file for the coNCePTuaL header file (‘`ncptl.h`’) and run-time library (‘`libncptl`’). This configuration file, ‘`ncptl.pc`’, should be installed in one of the directories searched by ‘`pkg-config`’ (‘`/usr/lib/pkgconfig`’ on some systems). Once ‘`ncptl.pc`’ is installed, ‘`pkg-config`’ can be used to compile C programs that require the coNCePTuaL header file and link programs that require the coNCePTuaL run-time library, as is shown in the following example:

```
cc ‘pkg-config --cflags ncptl‘ -c myprog.c
cc -o myprog myprog.o ‘pkg-config --libs ncptl‘
```

3 Usage

coNCePTuaL is more than just a language; it is a complete toolset which consists of the following components:

- the coNCePTuaL language (see Chapter 4 [Grammar], page 46)
- a compiler and run-time library for coNCePTuaL programs (see Section 3.1 [Compiling coNCePTuaL programs], page 11)
- a set of compiler backends that can generate code for a variety of languages and communication layers (see Section 3.2 [Supplied backends], page 13)
- utilities to help analyze the results (see Section 3.4.2 [logextract], page 28)

This chapter explains how to compile and run coNCePTuaL programs and how to interpret the log files they output.

3.1 Compiling coNCePTuaL programs

The coNCePTuaL compiler is called '`ncptl`' and is, by default, installed into '`/usr/local/bin`'. Executing *`ncptl --help`* produces a brief usage string:

```
Usage: ncptl [--backend=<string>] [--quiet] [--no-link | --no-compile]
         [--keep-ints] [--lenient] [--output=<file>]
         <file.ncptl> | --program=<program>
         [<backend-specific options>]
```

The usage string is followed by a list of installed backends.

The following list describes each compiler option in turn:

`--backend` (abbreviation: *`-b`*)
: Specify the module that coNCePTuaL should use as the compiler backend. '`ncptl`' must be told which backend to use with either *`--backend=`*`backend` or by setting the environment variable *`NCPTL_BACKEND`* to the desired backend. Running *`ncptl`* with no arguments (and without *`NCPTL_BACKEND`* set) will list the available backends. Most coNCePTuaL backends are code generators. For example, `c_mpi` causes '`ncptl`' to compile coNCePTuaL programs into C using MPI as the communication library. However, a backend need not generate code directly—or at all. The `c_trace` backend (see Section 3.2.4 [The c_trace backend], page 14), for instance, supplements the code generated by another backend by adding tracing output to it.

  '`ncptl`' searches for backends first using *`NCPTL_PATH`*, an environment variable containing a colon-separated list of directories (default: empty); then, in the directory in which coNCePTuaL installed all of its Python files; and finally, in the default Python search path. Non-directories (e.g., the '`.zip`' archives used in newer versions of Python) are not searched.

`--quiet` (abbreviation: *`-q`*)
: The *`--quiet`* option tells '`ncptl`' and the chosen backend to output minimal status information.

--no-link (abbreviation: *-c*)
: By default, '`ncptl`' instructs the backend to compile and link the user's coNCePTuaL program into an executable file. *--no-link* tells the backend to skip the linking step and produce only an object file.

--no-compile (abbreviation: *-E*)
: By default, '`ncptl`' instructs the backend to compile and link the user's coNCePTuaL program into an executable file. *--no-compile* tells the backend to skip both the compilation and the linking step and to produce only a source file in the target language.

--keep-ints (abbreviation: *-K*)
: coNCePTuaL backends normally delete any files created as part of the compiling or linking process. *--keep-ints* tells '`ncptl`' and the chosen backend to preserve their intermediate files.

--lenient (abbreviation: *-L*)
: The *--lenient* option tells the compiler to permit certain constructs that would otherwise result in a compilation error. First, using the same command-line option (either the long or short variant) for two different variables normally generates an '`Option` *opt* `is multiply defined`' error. (See Section 4.8.2 [Command-line arguments], page 86, for a description of how to declare command-line options in coNCePTuaL.) *--lenient* tells the coNCePTuaL compiler to automatically rename duplicate options to avoid conflicts. Only the option strings can be renamed; the programmer must still ensure that the option variables are unique. Second, using a variable without declaring it normally produces an error message at compile time. Passing *--lenient* to '`ncptl`' tells the compiler to automatically generate a command-line option for each missing variable. This is convenient when entering brief programs on the command line with *--program* (described below) as it can save a significant amount of typing.

--output (abbreviation: *-o*)
: '`ncptl`' normally writes its output to a file with the same base name as the input file (or '`a.out`' if the program was specified on the command line using *--program*). *--output* lets the user specify a file to which to write the generated code.

--program (abbreviation: *-p*)
: Because coNCePTuaL programs can be quite short *--program* enables a program to be specified in its entirety on the command line. The alternative to using *--program* is to specify the name of a file containing a coNCePTuaL program. By convention, coNCePTuaL programs have a '`.ncptl`' file extension.

The following—to be entered without line breaks—is a sample command line:

```
ncptl --backend=c_mpi --output=sample.c --program='Task 0 sends a 0
  byte message to task 1 then task 1 sends a 0 byte message to task 0
  then task 0 logs elapsed_usecs/2 as "Startup latency (usecs)".'
```

'ncptl' stops processing the command line at the first unrecognized option it encounters. That option and all subsequent options—including those which 'ncptl' would otherwise process—are passed to the backend without interpretation by 'ncptl'. Furthermore, the `--` (i.e., empty) option tells 'ncptl' explicitly to stop processing the command line at that point. For example, in the command *`ncptl --lenient myprogram.ncptl -- --help`*, 'ncptl' will process the *`--lenient`* option but will pass *`--help`* to the backend even though 'ncptl' has its own *`--help`* option.

3.2 Supplied backends

The coNCePTuaL 0.5.3 distribution includes the following compiler backends:

`c_seq` Generate ANSI C code with no communication support.

`c_mpi` Generate ANSI C code with calls to the MPI library for communication.

`c_udgram` Generate ANSI C code that communicates using Unix-domain (i.e., local to a single machine) datagram sockets.

`c_trace` Instrument a C-based backend either to include a call to `fprintf()` before every program event or to utilize the 'curses' library to display graphically the execution of a selected task.

`interpret`
Interpret a coNCePTuaL program, simulating any number of processors and checking for common problems such as deadlocks and mismatched sends and receives.

`dot` Output a program's parse tree in the Graphviz DOT format.

Each of these is described in turn in the following sections.

3.2.1 The c_seq backend

The `c_seq` backend is intended primarily to provide backend developers with a minimal C-based backend that can be used as a starting point for creating new backends. See Section 6.2 [Backend creation], page 94, explains how to write backends.

3.2.2 The c_mpi backend

The `c_mpi` backend is coNCePTuaL's workhorse. It generates parallel programs written in ANSI C that communicate using the industry-standard MPI messaging library.

By default, `c_mpi` produces an executable program that can be run with 'mpirun', 'prun', 'pdsh', or whatever other job-launching program is normally used to run MPI programs. When 'ncptl' is run with the *`--no-link`* option, `c_mpi` produces an object file that needs to be linked with the appropriate MPI library. When 'ncptl' is run with the *`--no-compile`* option, `c_mpi` outputs ANSI C code that must be both compiled and linked.

`c_mpi` honors the following environment variables when compiling and linking C+MPI programs: *`MPICC`*, *`MPICPPFLAGS`*, *`MPILDFLAGS`*, *`MPILIBS`*. If any of these variables is not found in the environment, `c_mpi` will use the value specified/discovered at configuration

time (see Section 2.1 [configure], page 4). *MPICC* defaults to the value of *CC*; the remaining variables are appended respectively to *CPPFLAGS*, *LDFLAGS*, and *LIBS*.

In the generated code, `MPI_Isend()` and `MPI_Irecv()` are used for asynchronous communication and `MPI_Send()` and `MPI_Recv()` are normally used for synchronous communication. However, the `c_mpi`-specific compiler option `--ssend` instructs `c_mpi` to replace all calls to `MPI_Send()` in the generated code with calls to `MPI_Ssend()`, MPI's synchronizing send function. A program's log files indicate whether the program was built to use `MPI_Send()` or `MPI_Ssend()`.

The following is a complete list of MPI functions employed by the `c_mpi` backend: `MPI_Allreduce()`, `MPI_Barrier()`, `MPI_Bcast()`, `MPI_Comm_rank()`, `MPI_Comm_size()`, `MPI_Comm_split()`, `MPI_Errhandler_set()`, `MPI_Finalize()`, `MPI_Init()`, `MPI_Irecv()`, `MPI_Isend()`, `MPI_Recv()`, `MPI_Send()`/`MPI_Ssend()`, `MPI_Waitall()`.

3.2.3 The c_udgram backend

coNCePTuaL program development on a workstation is facilitated by the `c_udgram` backend. `c_udgram` runs on only a single machine but, unlike `c_seq`, supports all of coNCePTuaL's communication statements. Communication is performed over Unix-domain datagram sockets. Unix-domain datagrams are reliable and guarantee order (unlike UDP/IP datagrams) but have a maximum packet size. `c_udgram` backend write this maximum to every log file and automatically packetizes larger messages.

By default, `c_udgram` produces an executable program that can be run directly from the command line. When '`ncptl`' is run with the `--no-link` option, `c_udgram` produces an object file that needs to be linked with the appropriate sockets library (on systems that require a separate library for socket calls). When '`ncptl`' is run with the `--no-compile` option, `c_udgram` outputs ANSI C code that must be both compiled and linked. Like all C-based backends, `c_udgram` honors the *CC*, *CPPFLAGS*, *LDFLAGS*, and *LIBS* environment variables when compiling and linking. Values not found in the environment are taken from those specified/discovered at configuration time (see Section 2.1 [configure], page 4).

In addition to supporting the default set of command-line options, programs generated using the `c_udgram` backend further support a `--tasks` option that designates the number of tasks to use:

```
-T, --tasks=<number>        Number of tasks to use [default: 1]
```

`c_udgram` programs spawn one OS-level process for each task in the program. They also create a number of sockets in the current directory named '`c_udgram_`⟨*tag*⟩'. These are automatically deleted if the program exits cleanly but will need to be removed manually in the case that the program is killed by an non-trappable signal.

3.2.4 The c_trace backend

While most coNCePTuaL backends are code generators, the `c_trace` backend adds tracing code to the code produced by a code-generating backend. `c_trace` is useful as a debugging aid and as a means to help understand the control flow of a coNCePTuaL program.

Command-line options for `c_trace`

When '`ncptl`' is passed `--backend=c_trace` as a command-line option, `c_trace` processes the following backend-specific command-line options:

`--trace=`*backend*
: Specify a backend that will produce C code for `c_trace` to trace. The `--trace` option is required to use `c_trace`; `c_trace` will issue an error message if `--trace` is not specified. The restrictions on *backend* are that it must produce C code and must be derived from the `c_generic` backend (see Section 6.2 [Backend creation], page 94). Improper backends cause `c_trace` to abort abnormally.

`--curses`
: Instead of injecting `fprintf()` statements into the generated C code, inject calls to the '`curses`' (or '`ncurses`') library to show graphically the line of code currently executing on a given processor.

Default `c_trace` tracing

Without `--curses`, `c_trace` alters the generated C code to write data like the following to the standard error device:

```
[TRACE] phys: 1 | virt: 1 | action: RECV | event: 1 / 44001 | lines: 18 - 18
[TRACE] phys: 0 | virt: 0 | action: RESET | event: 1 / 88023 | lines: 17 - 17
[TRACE] phys: 0 | virt: 0 | action: SEND | event: 2 / 88023 | lines: 18 - 18
[TRACE] phys: 0 | virt: 0 | action: RECV | event: 3 / 88023 | lines: 19 - 19
[TRACE] phys: 1 | virt: 1 | action: SEND | event: 2 / 44001 | lines: 19 - 19
[TRACE] phys: 1 | virt: 1 | action: RECV | event: 3 / 44001 | lines: 18 - 18
[TRACE] phys: 0 | virt: 0 | action: CODE | event: 4 / 88023 | lines: 20 - 21
[TRACE] phys: 0 | virt: 0 | action: RESET | event: 5 / 88023 | lines: 17 - 17
[TRACE] phys: 0 | virt: 0 | action: SEND | event: 6 / 88023 | lines: 18 - 18
[TRACE] phys: 0 | virt: 0 | action: RECV | event: 7 / 88023 | lines: 19 - 19
```

⋮

The format is designed to be easy to read and easy for a program to parse. Each line of trace data begins with the string '`[TRACE]`' and lists the (physical) processor number, the (virtual) task ID, the action (a.k.a., event type) that is about to be performed, the current event number and total number of events that will execute on the given processor, and the range of lines of source code to which the current event corresponds. An "event" corresponds more-or-less to a statement in the coNCePTuaL language.[1] Loops are unrolled at initialization time and therefore produce no events. Section 6.2.3 [Generated code], page 98, lists and briefly describes the various event types.

[1] A more precise correspondence is to a ⟨*simple_stmt*⟩ in the formal grammar presented in Chapter 4 [Grammar], page 46.

`c_trace` tracing with 'curses'

The *--curses* option enables a more interactive tracing environment. Generated programs must be linked with the 'curses' (or compatible, such as 'ncurses') library. The resulting executable supports the following additional command-line options:

-D, *--delay=number*
: delay in milliseconds after each screen update ('0'=no delay)

-M, *--monitor=number*
: processor number to monitor

-B, *--breakpoint=number*
: source line at which to enter single-stepping mode ('-1'=none; '0'=first event)

When the program is run it brings up a screen like the following:

```
 1.  # Determine computational "noise"
 2.
 3.  Require language version "0.5.2a".
 4.
 5.  accesses is "Number of data accesses to perform" and comes from
 6.    "--accesses" or "-a" with default 500000.
 7.
 8.  trials is "Number of timings to take" and comes from "--timings" or
 9.    "-t" with default 1000.
10.
11.  For trials repetitions {
12.    all tasks reset their counters then
13.    all tasks touch a 1 word memory region accesses times with stride 0 w
14.    all tasks log a histogram of elapsed_usecs as "Actual time (usecs)"
15.  }


Phys: 0  Virt: 0  Action: RESET    Event:    1/3001
```

The program displays its source code (truncated vertically if too tall and truncated horizontally if too wide) at the top of the screen and a status bar at the bottom of the screen. As the program executes, a cursor indicates the line of source code that is currently executing. Likewise, the status bar updates dynamically to indicate the processor's current task ID, action, and event number. In 'curses' mode, the program's standard output (see Section 4.5.2 [Writing to standard output], page 71) is suppressed so as not to disrupt the trace display.

Programs traced with `c_trace` and the *--curses* option are made interactive and support the following (case-insensitive) keyboard commands:

'S'
: Enable single-stepping mode. While single-stepping mode is enabled the traced processor will execute only one event per keystroke from the user.

'*space*'
: Disable single-stepping mode. The program executes without further user intervention.

‘D’ Delete the breakpoint.

‘Q’ Quit the program. The log file will indicate that the program did not run to completion.

All other keystrokes cause the program to advance to the next event immediately.

A single breakpoint can be set using the program’s *-B* or *--breakpoint* command-line option. Whenever the monitored processor reaches the source-code line at which a breakpoint has been set, it enters single-stepping mode exactly as if *S* were pressed. Setting a breakpoint at line 0 tells the program to begin single-stepping as soon as the program begins. Note that only lines corresponding to coNCePTuaL events can support breakpoints.

Offline tracing with ‘curses’

The `c_trace` backend can be told to trace by writing messages to the standard error device or by employing an interactive display. These two alternatives can be combined to support offline tracing of a coNCePTuaL program. The idea is to compile the program without the *--curses* option. When running the program, the standard-error output should be redirected to a file. The ‘`replaytrace`’ utility, which comes with coNCePTuaL, can then be used to play back the program’s execution by reading and displaying the file of redirected trace data.

‘`replaytrace`’ accepts the following command-line options, which correspond closely to those accepted by a program compiled with the *--curses* option to `c_trace`:

--trace=file
Specify a file containing redirected trace data. *file* defaults to the standard input device.

--delay=number
Specify the delay in milliseconds after each screen update (‘0’=no delay).

--monitor=processor
Specify the processor number to monitor. *processor* defaults to ‘0’.

--breakpoint=line
Specify a line of source code at which to enter single-stepping mode (‘-1’=none; ‘0’=first event).

In addition, ‘`replaytrace`’ requires that the coNCePTuaL source-code file be specified on the command line, as the source code is not included in the trace data.

The interactive display presented by the offline ‘`replaytrace`’ tool is nearly identical to that presented by a program compiled with the *--curses* option to `c_trace`.

3.2.5 The interpret backend

Like the `c_udgram` backend (see Section 3.2.3 [The c_udgram backend], page 14), the `interpret` backend is designed to help programmers test coNCePTuaL code. The `interpret` backend does not output code. As its name implies, `interpret` is an *interpreter* of coNCePTuaL programs rather than a compiler. `interpret` exhibits the following salient features:

1. Some programs run faster than with a compiler because the interpreter does not actually send messages. `interpret` merely simulates communication.
2. `interpret` can simulate massively parallel computer systems from a single process.
3. As `interpret` runs it checks for common communication errors such as deadlocks, asynchronous sends and receives that are never completed, and blocking operations left over at the end of the program (which would cause hung tasks).

The drawbacks are that `interpret` is slow when interpreting control-intensive programs and that timing measurements are not indicative of any real network. `interpret` is intended primarily as a development tool for helping ensure the correctness of coNCePTuaL programs.

The `interpret` backend accepts all of the command-line options described in Section 3.3 [Running coNCePTuaL programs], page 21, plus the following two options:

```
-O, --onlylog=<string>      Comma-separated list of processor ranges that
                            are allowed to write log files [default:
                            "0-0"]
-T, --tasks=<number>        Number of tasks to use [default: 1]
```

The *--tasks* option specifies the number of tasks to simulate. Because this number can be quite large the *--onlylog* option limits the set of processors that are allowed to create log files. That way, if task 0 is the only task out of thousands that logs any data, '`--onlylog=0`' ensures that only one log file will be produced, not thousands. By default, all processors create a log file.

All other command-line arguments are passed to the program being interpreted.

As an example of the `interpret` backend's usage, here's how to simulate 100,000 processors communicating in a simple ring pattern:

```
% ncptl --backend=interpret --lenient --program='All tasks t send
    nummsgs 1024 gigabyte messages to task t+1 then task num_tasks-1
    sends nummsgs 1024 gigabyte messages to task 0.' --tasks=100000
    --nummsgs=5
```

The preceding command ran to completion in under 3 minutes on a 1.5 GHz Xeon uniprocessor workstation—not too bad considering that 488 petabytes of data are transmitted serially on the program's critical path.

The `interpret` backend is especially useful for finding communication-related program errors:

```
% ncptl --backend=interpret --quiet --program='All tasks t send
    a 10 doubleword message to task (t+1) mod num_tasks.' --tasks=3
<command line>: The following tasks have deadlocked: 0 --> 2 --> 1
    --> 0
```

Deadlocked tasks are shown with '`-->`' signifying "is blocked waiting for". In the preceding example, all receives are posted before all sends. Hence, task 0 is blocked waiting for task 2 to send it a message. Task 2, in turn, is blocked waiting for task 1 to sent it a message. Finally, task 1 is blocked waiting for task 0 to send it a message, which creates a cycle of dependencies.

The `interpret` backend can find other errors, as well:

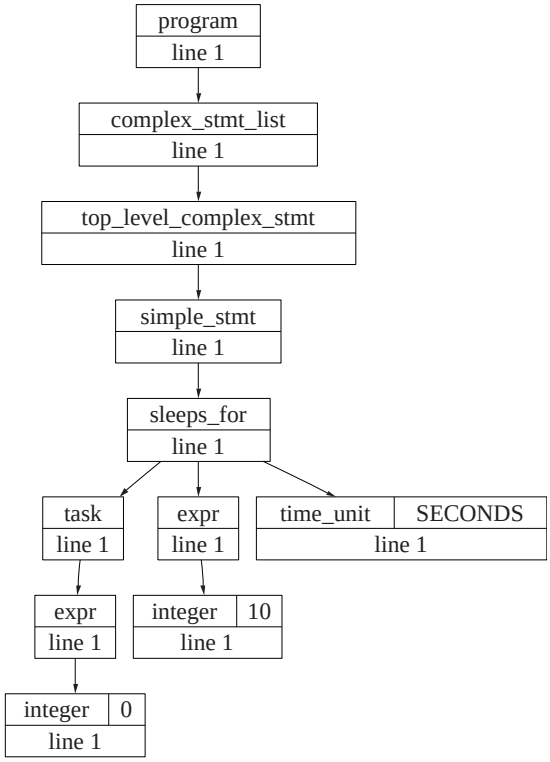
```
% ncptl --backend=interpret --quiet --program='All tasks t
    asynchronously send a 10 doubleword message to task (t+1) mod
    num_tasks.' --tasks=4
<command line>: The program ended with the following leftover-event
    errors:
   * Task 0 posted a asynchronous SEND which was never waited for
   * Task 0 posted a asynchronous RECEIVE which was never waited for
   * Task 1 posted a asynchronous SEND which was never waited for
   * Task 1 posted a asynchronous RECEIVE which was never waited for
   * Task 2 posted a asynchronous SEND which was never waited for
   * Task 2 posted a asynchronous RECEIVE which was never waited for
   * Task 3 posted a asynchronous SEND which was never waited for
   * Task 3 posted a asynchronous RECEIVE which was never waited for
```

```
% ncptl --backend=interpret --quiet --program='Task 0 sends a 40
    kilobyte message to unsuspecting task 1 then task 0 receives a 40
    kilobyte message from task 1.' --tasks=2
<command line>: The program ended with the following leftover-event
    errors:
   * Task 0 sent to task 1 a message which was never received
   * The program ended with task 0 blocked on a RECEIVE
```

In short, it is well worth testing the correctness of new coNCePTuaL programs with `interpret` before performing timing runs with one of the message-passing backends.

3.2.6 The dot backend

The `dot` backend generates code, but not executable code. DOT (described in `http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`) is a format for describing graphs in terms of their edges and vertices. The tools in the Graphviz suite (available from `http://www.research.att.com/sw/tools/graphviz/`) typeset DOT files in a variety of output formats and using a variety of graph-layout algorithms. coNCePTuaL's `dot` backend outputs in DOT format the abstract-syntax tree corresponding to a given coNCePTuaL program. As an example, `dot` renders the one-line coNCePTuaL program '`TASK 0 SLEEPS FOR 10 SECONDS.`' as follows:



`dot` is expected to be of particular use to backend developers, who can use it to help prioritize the methods that need to be implemented (i.e., implementing first the AST node types needed by in a trivial program, then those needed by successively more complex programs).

The `dot` backend accepts the following options from the '`ncptl`' command line:

`--format=`dot_format

The programs in the Graphviz suite can output graphs in a variety of formats such as PostScript, SVG, and PNG. By default, the `dot` backend outputs PostScript. The *`--format`* option specifies an alternate format to use. At the time of this writing, the Graphviz programs support the following formats: `canon`, `cmap`, `dot`, `fig`, `gd`, `gd2`, `gif`, `hpgl`, `imap`, `ismap`, `jpeg`, `jpg`, `mif`, `mp`, `pcl`, `pic`, `plain`, `plain-ext`, `png`, `ps`, `ps2`, `svg`, `svgz`, `vrml`, `vtx`, `wbmp`, and `xdot`. See the Graphviz documentation for more information about these formats.

`--extra-dot=`dot_code

The `dot` backend's *`--extra-dot`* option enables the user to inject arbitrary DOT code into the generated file. For example, specifying *`--extra-dot="node [shape=Mrecord]"`*[2] tells DOT to use draw nodes as rounded rectangles and specifying *`--extra-dot='edge [color="green"]'`* colors all edges green. *`--extra-dot`* can be specified repeatedly on the command line; `dot` concatenates all of the extra DOT code with intervening semicolons.

[2] `dot` automatically places a semicolon after the extra DOT code.

--no-lines
: By default, each AST node indicates the lines in the program's source code to which it corresponds. The *--no-lines* option suppresses the outputting of source-code line numbers.

--no-attrs
: Every node in the AST has a type. Some nodes additionally have an attribute. `dot` normally outputs attributes but *--no-attrs* prevents `dot` from doing so.

The *DOT* environment variable names the Graphviz program that `dot` should run on the generated code. If *DOT* is not set, `dot` uses whatever value was specified/discovered at configuration time (see Section 2.1 [configure], page 4), with the default being '`dot`'. By default, `dot` produces DOT code and runs this through the designated Graphviz program to produce a PostScript file (or whatever format is named by the *--format* option). If '`ncptl`' is run with either the *--no-link* or *--no-compile* options, it produces a DOT file that should be run manually through '`dot`' or another Graphviz tool.

3.3 Running coNCePTuaL programs

coNCePTuaL programs can be run like any other program built with the same compiler and communication library. For example if a program '`myprog`' was built with coNCePTuaL's C+MPI backend, the program might be run with a command like *mpirun -np nodes myprog* or *prun -Nnodes myprog* or *pdsh -w node_list myprog*. The important point is that job launching is external to coNCePTuaL. A coNCePTuaL program is oblivious to whether it is being run with a single thread on each multiprocessor node or with one thread on each CPU, for example. However, coNCePTuaL log files do include the host name in the header comments (see Section 3.4.1 [Log-file format], page 23) so job-launching parameters can potentially be inferred from those.

coNCePTuaL programs automatically support a "help" option. This is usually specified as *--help*, *-h*, or *-?*, depending on which option-parsing library '`configure`' configured in. The output of running *myprog --help* most likely looks something like this:

```
Usage: myprog [OPTION...]
  -C, --comment=<string>      Additional commentary to write to the log
                              file, @FILE to import commentary from FILE,
                              or !COMMAND to import commentary from COMMAND
                              (may be specified repeatedly)
  -L, --logfile=<string>      Log file template [default: "a.out-%p.log"]
  -N, --no-trap=<string>      List of signals that should not be trapped
                              [default: ""]
  -S, --seed=<number>         Seed for the random-number generator
                              [default: 0]
  -W, --watchdog=<number>     Number of minutes after which to kill the job
                              (-1=never) [default: -1]

Help options:
  -?, --help                  Show this help message
  --usage                     Display brief usage message
```

Although a coNCePTuaL program can specify its own command-line options (see Section 4.8.2 [Command-line arguments], page 86), a few are provided by default. In addition to *--help* these include *--comment*, *--logfile*, *--no-trap*, *--seed*, and *--watchdog*:

--comment

--comment makes it possible to add arbitrary commentary to a log file. This is useful for incorporating information that coNCePTuaL would be unable to (or simply does not currently) determine on its own, for example, *--comment="Last experiment before upgrading the network device driver"*. Two special cases are supported:

1. If the comment string begins with '`@`' then the remainder of the string is treated as a filename. Each line of the corresponding file is treated as a separate comment string. Hence, if the file '`sysdesc.txt`' contains the lines '`Using FooBarNet`' and '`Quux is enabled`', then specifying *--comment=sysdesc.txt* is equivalent to specifying both *--comment="Using FooBarNet"* and *--comment="Quux is enabled"*.
2. If the comment string begins with '`!`' then the remainder of the string is treated as a shell command. The command is executed and each line of its output is treated as a separate comment string. For example, *--comment='!lspci | grep -i net'* executes '`lspci`', extracts only those lines containing the string '`net`', and makes log-file comments out of the result. Note that *--comment='!command'* differs from *--comment="`command`"* in that the former causes *command* to be executed individually by each process in the program while the latter executes *command* only once and only before launching the program. Also note that '`!`' must be escaped in '`csh`' and derivitive shells (i.e., *--comment='\!command'*).

--logfile

--logfile specifies a template for naming log files. Each task maintains a log file based on the template name but with '`%p`' replaced with the processor number, '`%r`' replaced with the run number (the smallest nonnegative integer that produces a filename which does not already exist), and '`%%`' replaced with a literal "%" character. The program outputs an error message and aborts if the log-file template does not contain at least one '`%p`'. The only exception is that an empty template (i.e., *--logfile=""*) inhibits the production of log files entirely.

--no-trap

--no-trap specifies a list of signals or ranges of signals that should not be trapped. For example, *--no-trap=10-12,17* prevents signals 10, 11, 12, and 17 from being trapped.[3] Signals can also be referred to by name, with or without a '`SIG`' prefix. Also, names and numbers can be freely mixed. Hence, *--no-trap=10-12,INT,17,SIGSTOP,SIGCONT* is a valid argument to a coNCePTuaL program. Because signal reception can adversely affect performance, coNCePTuaL's default behavior is to terminate the program on receipt of a

[3] On some platforms, these signals correspond to `SIGUSR1`, `SIGSEGV`, `SIGUSR2`, and `SIGCHLD`, respectively.

signal. However, some signals may be necessary for the underlying communication layer's proper operation. *--no-trap* enables such signals to pass through coNCePTuaL untouched. (Some signals, however, are needed by coNCePTuaL or by a particular backend and are always trapped.)

--seed *--seed* (which selects a different default value on each run) is used in any program that utilizes the `RANDOM TASK` construct (see Section 4.7.3 [Binding variables], page 83) or that sends message `WITH VERIFICATION` (see Section 4.4.1 [Message specifications], page 63).

--watchdog

--watchdog is useful when batch-submitting a sequence of coNCePTuaL jobs as it prevents a hung program (caused, for example, by data corruption within the messaging layer, deadlock within the program's communication pattern, or an undetected error in the network hardware) from preventing the remaining jobs from running.

3.4 Interpreting coNCePTuaL log files

Any coNCePTuaL program that uses the `LOGS` keyword (see Section 4.5.3 [Writing to a log file], page 71) will produce a log file as it runs. The coNCePTuaL run-time library writes log files in a simple, plain-text format. In addition to measurement data, a wealth of information is stored within log-file comments. coNCePTuaL comes with a tool, '`logextract`', which can extract data and other information from a log file and convert it into any of a variety of other formats.

3.4.1 Log-file format

The coNCePTuaL run-time library writes log files in the following (textual) format:

- Lines beginning with '`#`' are comments.
- Columns are separated by commas.
- Strings are output between double quotes. Literal double-quotes are output as '`\"`' and literal backslashes are output as '`\\`'.

A sample log file is listed below. The log file is presented in its entirety.

```
###############################################################################
# ==================
# coNCePTuaL log file
# ==================
# coNCePTuaL version: 0.5.2
# coNCePTuaL backend: c_mpi (C + MPI)
# Executable name: /home/pakin/src/coNCePTuaL/example
# Working directory: /home/pakin/src/coNCePTuaL
# Command line: example
# Number of tasks: 2
# Processor (0<=P<tasks): 0
# Host name: a1
# Operating system: Linux 2.4.21-3.5qsnet #2 SMP Thu Aug 7 10:51:04 MDT 2003
```

```
# CPU vendor: GenuineIntel
# CPU architecture: ia64
# CPU model: 1
# CPU count: 2
# CPU frequency: 1300000000 Hz (1.3 GHz)
# Cycle-counter frequency: 1300000000 Hz (1.3 GHz)
# OS page size: 16384 bytes
# Physical memory: 2047901696 bytes (1.9 GB)
# Library compiler+linker: /usr/bin/gcc
# Library compiler options: -Wall -W -g -O3
# Library linker options: -lpapi -lm -lpopt
# Library compiler mode: LP64
# Dynamic libraries used: /usr/local/lib/libpapi.so /lib/libm-2.2.4.so /usr/lib/libpop
# Microsecond timer type: inline assembly code
# Average microsecond timer overhead: <1 microsecond
# Microsecond timer increment: 1.00986 +/- 0.298216 microseconds (ideal: 1 +/- 0)
# Minimum sleep time: 1952.44 +/- 2.51794 microseconds (ideal: 1 +/- 0)
# WARNING: Sleeping exhibits poor granularity (not a serious problem).
# WARNING: Sleeping has a large error component (not a serious problem).
# Process CPU timer: getrusage()
# Process CPU-time increment: 976.55 +/- 0.5 microseconds (ideal: 1 +/- 0)
# WARNING: Process timer exhibits poor granularity (not a serious problem).
# Log file template: example-%p.log
# Number of minutes after which to kill the job (-1=never): -1
# List of signals that should not be trapped: 14
# MPI send routine: MPI_Send()
# Compilation command line: /usr/lib/mpi/mpi_gnu/bin/mpicc -I/tmp/ncptl/include  -I/us
# Log creator: Scott Pakin
# Log creation time: Mon Sep 13 19:12:49 2004
#
# Environment variables
# ---------------------
# CVS_RSH: /usr/bin/ssh
# DISPLAY: localhost:10.0
# DYNINSTAPI_RT_LIB: /home/pakin/dyninstAPI-3.0/lib/i386-unknown-linux2.2/libdyninstAP
# DYNINST_ROOT: /home/pakin/dyninstAPI-3.0
# EDITOR: /usr/bin/emacs
# GROUP: CCS3
# HOME: /home/pakin
# HOST: a0
# HOSTNAME: a0
# HOSTTYPE: unknown
# KDEDIR: /usr
# LANG: en_US
# LD_LIBRARY_PATH: /users/pakin/lib:/usr/lib:/usr/ccs/lib:/opt/SUNWspro/lib:/usr/dt/li
# LESSOPEN: |/usr/bin/lesspipe.sh %s
# LOGNAME: pakin
# LPDEST: lwy
# LS_COLORS: no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:o
```

```
# MACHTYPE: unknown
# MAIL: /var/mail/pakin
# MANPATH: /usr/man:/opt/SUNWspro/man:/usr/dt/man:/usr/openwin/man:/usr/X11R6/man:/usr
# MOZILLA_HOME: /usr/local/netscape/java
# NAME: Scott Pakin
# ORGANIZATION: Los Alamos National Lab
# OSTYPE: linux
# PATH: .:/home/pakin/bin:/usr/local/bin:/usr/dt/bin:/usr/openwin/bin:/usr/X11R6/bin:/
# PRINTER: lwy
# PVM_ROOT: /usr/share/pvm3
# PVM_RSH: /usr/bin/rsh
# PWD: /home/pakin/src/coNCePTuaL
# QTDIR: /usr/lib/qt-2.3.1
# REMOTEHOST: antero.c3.lanl.gov
# RMS_JOBID: 24286
# RMS_MACHINE: a
# RMS_NNODES: 2
# RMS_NODEID: 0
# RMS_NPROCS: 2
# RMS_PROCID: 0
# RMS_RANK: 0
# RMS_RESOURCEID: parallel.25183
# RMS_STOPONELANINIT: 0
# SHELL: /bin/tcsh
# SHLVL: 2
# SSH_AGENT_PID: 24930
# SSH_ASKPASS: /usr/libexec/openssh/gnome-ssh-askpass
# SSH_AUTH_SOCK: /tmp/ssh-XXByeFZc/agent.24905
# SSH_CLIENT: 128.165.20.177 34362 22
# SSH_TTY: /dev/pts/0
# SUPPORTED: en_US:en
# TERM: xterm
# TZ: MST7MDT
# USER: pakin
# VENDOR: unknown
#
# coNCePTuaL source code
# ----------------------
#     FOR 10 REPETITIONS {
#       TASK 0 RESETS ITS COUNTERS THEN
#       TASK 0 SENDS A 0 BYTE MESSAGE TO TASK 1 THEN
#       TASK 1 SENDS A 0 BYTE MESSAGE TO TASK 0 THEN
#       TASK 0 LOGS EACH elapsed_usecs/2 AS "Latency (usecs)"
#     }
#
###############################################################################
"Latency (usecs)"
"(all data)"
194.5
```

```
5.5
5.5
5
5
5
5.5
5
5
5
################################################################################
# Program exited normally.
# Log completion time: Mon Sep 13 19:12:49 2004
# Elapsed time: 0 seconds
# Process CPU usage (user+system): 0 seconds
# Task IDs assigned to processor 0: 0
# Processors assigned to task ID 0: 0
################################################################################
```

As the preceding example indicates, a log file's comment block can be divided into multiple stanzas:

- a list of ⟨*key*:*value*⟩ pairs that describe various characteristics of the run-time environment, including hardware and software identification, timer quality, values of command-line arguments, and a timestamp
- a dump of the environment variables active when the program ran
- the complete coNCePTuaL source program

Two rows of headers and the measurement data follow the comment block and a brief trailer comment completes the log file.

The motivation for writing so much information to the log file is to facilitate reproduction of the experiment. The ideal situation is for a third party to be able to look at a coNCePTuaL log file and from that, recreate the original experiment and get identical results.

Some of the comments that may benefit from additional explanation include the following:

'`Library compiler mode`'
: '`LP64`' means "**L**ong integers and **P**ointers contain exactly **64** bits while ordinary integers contain exactly 32 bits". '`ILP32`' means "ordinary **I**ntegers, **L**ong integers, and **P**ointers all contain exactly **32** bits". The library compiler mode will be '`nonstandard`' for any other combination of datatype sizes.

'`Average timer overhead`'
: During initialization, the coNCePTuaL run-time library performs some calibration routines. Among these routines is a measurement of the quality of whatever mechanisms the library is using to measure elapsed time. In the sample log file presented above, the mechanism used was '`inline assembly code`', meaning the run-time library reading the hardware cycle counter without going

through a standard library or system call. Some alternative mechanisms include `get_cycles()`, `PAPI_get_real_usec()`, `clock_gettime(CLOCK_SGI_CYCLE)`, `clock_gettime(CLOCK_REALTIME)`, and `gettimeofday()`. Not every platform supports every mechanism; the coNCePTuaL '`configure`' script selects the "best" mechanism (determined statically) from those available.

The log file then reports "average timer overhead" as the mean time between back-to-back invocations of whichever timer routine is being used. Ideally, the mean should be '`<1 microsecond`' but this is not the case on all systems. Large values indicate that a performance penalty is charged every time a coNCePTuaL program reads the timer.

'`Timer increment`'

In addition to measuring the overhead of reading the timer, the coNCePTuaL run-time library also measures timer accuracy. The library expects to be able to read the timer with microsecond accuracy. That is, the time reported should not increase by more than a microsecond between successive readings of the timer. To gauge timer accuracy, the library's initialization routine performs a number of back-to-back invocations of the timer routine and reports the mean and standard deviation of the number of microseconds that elapsed between readings, discarding any deltas of zero microseconds. Ideally, the microsecond timer, when read multiple times in rapid succession, should report nonzero increments of exactly one microsecond with no variation. The log file will contain warning messages if the increment or standard deviation are excessively large, as this may indicate a large margin of error in the measurement data.

'`Process CPU-time increment`'
'`Process CPU usage (user+system)`'

Log files end with a trailer section that includes '`Process CPU usage (user+system)`', which indicates the subset of total wall-clock time for which the program was running ('`user`') or for which the operating system was running on the program's behalf ('`system`'). The log-file header reports as '`Process CPU-time increment`' the resolution of the timer user to report process CPU time. Note that process CPU time is not exported to coNCePTuaL programs; it is therefore much less critical than the wall-clock timer and is reported primarily for informational purposes.

'`Task IDs assigned to processor` ⟨*number*⟩'
'`Processors assigned to task ID` ⟨*number*⟩'

coNCePTuaL distinguishes between "processors" and "tasks". A "processor" typically corresponds to a physical CPU but may be virtualized as a thread or a process by the underlying communication layer. Each processor is referred to by a unique number from 0 up to (but not including) the total number of processors available to the program. Processor IDs are assigned by the underlying communication layer so they may actually be MPI ranks in an MPI program or thread IDs in an OpenMP program. Because job launching is external to coNCePTuaL, a program has no knowledge of how processor IDs are mapped to physical processors. Processor IDs do not change over the lifetime of the program. coNCePTuaL programs do not generally refer to processors but

rather to tasks. A task ID is like a processor in that it is a unique number from 0 up to (but not including) the number of processors in the program. However, coNCePTuaL programs have explicit control over the assignment of task IDs to processors and can alter the mapping at any point in the program (see Section 4.6.5 [Reordering task IDs], page 76). When a coNCePTuaL program begins, processor IDs and task IDs are equal.

As a debugging aid, each task that writes a log file outputs as '`Task IDs assigned to processor` ⟨*number*⟩' the task IDs that were assigned to it during the course of the program's execution. However, showing task IDs from the perspective of a given processor can be confusing, as coNCePTuaL programs express processors from the perspective of a given task ID. Each processor therefore additionally outputs '`Processors assigned to task ID` ⟨*number*⟩' which shows the more intuitive task ID→processor mapping. Note, however, that the process that outputs '`Processors assigned to task ID` ⟨*number*⟩' is the process that *originally* had task ID ⟨*number*⟩; the task ID may have been changed during the program's execution.

3.4.2 'logextract'

To facilitate converting coNCePTuaL log files into input data for other applications, coNCePTuaL provides a Perl script called '`logextract`'. '`logextract`' can extract the data from a log file—as well as various information that appears in the log file's comments—into a variety of formats suitable for graphing or typesetting.

Running *`logextract --usage`* causes '`logextract`' to list a synopsis of its core command-line options to the standard output device; running *`logextract --help`* produces basic usage information; and, running *`logextract --man`* outputs a complete manual page. The following pages show the '`logextract`' documentation as produced by *`logextract --man`*.

NAME

logextract - Extract various bits of information from a coNCePTuaL log file

SYNOPSIS

logextract *--usage | --help | --man*

logextract [*--extract*=[data|params|source|warnings]] [*--format*=*format*] [*format-specific options...*] [*--before*=*string*] [*--after*=*string*] [*--force-merge*[=*number*]] [*--output*=*filename*] [*filename...*]

DESCRIPTION

Background coNCePTuaL is a domain-specific programming language designed to facilitate writing networking benchmarks and validation suites. coNCePTuaL programs can log data to a file but in only a single file format. '`logextract`' extracts this log data and outputs it in a variety of formats for use with other applications.

The coNCePTuaL-generated log files that serve as input to '`logextract`' are plain ASCII files. Syntactically, they contain a number of newline-separated tables. Each table contains a number of newline-separated rows of comma-separated columns. This is known generically as *comma-separated value* or *CSV* format. Each table begins with two rows of header text followed by one or more rows of numbers. Text is written within double quotes. Double-quote characters and backslashes within text are escaped with a backslash. No other escaped characters are recognized. Lines that begin with `#` are considered comments.

Semantically, there are four types of data present in every coNCePTuaL-generated log file:

1. The complete source code of the coNCePTuaL program that produced the log file
2. Characteristics of the run-time environment and the values of all command-line parameters
3. A list of warning messages that coNCePTuaL issued while analyzing the run-time environment
4. One or more tables of measurement data produced by the coNCePTuaL program

The first three items appear within comment lines. The measurement data is written in CSV format.

Extracting information from coNCePTuaL log files It is common to want to extract information (especially measurement data) from log files. For simple formatting operations, a one-line awk or Perl script suffices. However, as the complexity of the formatting increases, the complexity of these scripts increases even more. That's where '`logextract`' fits in. '`logextract`' makes it easy to extract any of the four types of log data described above and format it in variety of ways. Although the number of options that '`logextract`' supports may be somewhat daunting, it is well worth learning how to use '`logextract`' to avoid reinventing the wheel every time a coNCePTuaL log file needs to be processed. '`logextract`' takes care of all sorts of special cases that crop up when manipulating coNCePTuaL log files.

OPTIONS

'`logextract`' accepts the following command-line options regardless of what data is extracted from the log file and what formatting occurs:

`-h`, `--help`
: Output the Synopsis section and the Options section, then exit the program.

`-m`, `--man`
: Output a complete Unix man ("manual") page for '`logextract`', then exit the program.

`-e` *info*, `--extract`=*info*
: Specify what sort of data should be extracted from the log file. Acceptable values for *info* are listed and described in the Additional Options section and include `data`, `params`, `env`, and `source`.

`-f` *format*, `--format`=*format*
: Specify how the extracted data should be formatted. Valid arguments depend upon the value passed to `--extract` and include such formats as `csv`, `html`, `latex`, `text`, and `bash`. See the Additional Options section for details, explanations, and descriptions of applicability.

`-b` *string*, `--before`=*string*
: Output an arbitrary string of text before any other output. *string* can contain escape characters such as `\n` for newline, `\t` for tab, and `\\` for backslash.

`-a` *string*, `--after`=*string*
: Output an arbitrary string of text after all other output. *string* can contain escape characters such as `\n` for newline, `\t` for tab, and `\\` for backslash.

`-F` [*number*], `--force-merge`[=*number*]
: Try extra hard to merge multiple log files, even if they seem to have been produced by different programs or in different execution environments. This generally implies padding empty rows and columns with blanks. However, if `--force-merge` is given a numeric argument, the value of that argument is used instead of blanks to pad empty locations. Note that `--force-merge` is different from `--force-merge=0` because data-merging functions (`mean`, `max`, etc.) ignore blanks but consider zeroes.

`--output`=*filename*
: '`logextract`' normally writes to the standard output device. The `--output` option redirects '`logextract`''s output to a file.

The above is merely a terse summary of '`logextract`''s command-line options. The reader is directed to the Additional Options section for descriptions of the numerous ways that '`logextract`' can format information. Note that `--extract` and `--format` are the two most common options as they specify what to extract and how to format it; most of the remaining options in the Additional Options section exist to provide precise control over formatting details.

ADDITIONAL OPTIONS

'`logextract`''s command-line options follow a hierarchy. At the top level is `--extract`, which specifies which of the four types of data '`logextract`' should extract. Next, `--format`

specifies how the extracted data should be formatted. Valid values for `--format` differ based on the argument to `--extract`. Finally, there are various format-specific options that fine-tune the formatted output. Each output format accepts a different set of options. Many of the options appear at multiple places within the hierarchy, although usually with different default values.

The following hierarchical list describes all of the valid combinations of `--extract`, `--format`, and the various format-specific options:

`--extract=data` [default]
: Extract measurement data

  `--format=csv` [default]
  : Output each table in comma-separated-value format

    `--noheaders`
    : Do not output column headers

    `--colbegin=`*string*
    : Override the text placed at the beginning of each data column [default: “”]

    `--colsep=`*string*
    : Override the text used to separate data columns [default: “,”]

    `--colend=`*string*
    : Override the text placed at the end of each data column [default: “”]

    `--rowbegin=`*string*
    : Override the text placed at the beginning of each data row [default: “”]

    `--rowsep=`*string*
    : Override the text used to separate data rows [default: “”]

    `--rowend=`*string*
    : Override the text placed at the end of each data row [default: “\\n”]

    `--hcolbegin=`*string*
    : Override the text placed at the beginning of each header column [default: same as `colbegin`]

    `--hcolsep=`*string*
    : Override the text used to separate header columns [default: same as `colsep`]

    `--hcolend=`*string*
    : Override the text placed at the end of each header column [default: same as `colend`]

`--hrowbegin=`*string*
: Override the text placed at the beginning of each header row [default: same as `rowbegin`]

`--hrowsep=`*string*
: Override the text used to separate header rows [default: same as `rowsep`]

`--hrowend=`*string*
: Override the text placed at the end of each header row [default: same as `rowend`]

`--tablebegin=`*string*
: Override the text placed at the beginning of each table [default: “”]

`--tablesep=`*string*
: Override the text used to separate tables [default: “”]

`--tableend=`*string*
: Override the text placed at the end of each table [default: “`\\n`”]

`--quote=`*string*
: Override the text used to begin quoted text [default: “`"`”]

`--unquote=`*string*
: Override the text used to end quoted text [default: same as `quote`]

`--excel`
: Output strings in a format readable by Microsoft Excel

`--merge=`*function*
: Specify how to merge data from multiple files [default: “`mean`”]

`--showfnames=`*option*
: Add an extra header row showing the filename the data came from [default: “`none`”]

`--format=tsv`
: Output each table in tab-separated-value format

`--noheaders`
: Do not output column headers

`--colbegin=`*string*
: Override the text placed at the beginning of each data column [default: “”]

`--colsep=`*string*
: Override the text used to separate data columns [default: “`\\t`”]

`--colend=`*string*
: Override the text placed at the end of each data column [default: “”]

`--rowbegin=`*string*
: Override the text placed at the beginning of each data row [default: “”]

`--rowsep=`*string*
: Override the text used to separate data rows [default: “”]

`--rowend=`*string*
: Override the text placed at the end of each data row [default: “`\\n`”]

`--hcolbegin=`*string*
: Override the text placed at the beginning of each header column [default: same as `colbegin`]

`--hcolsep=`*string*
: Override the text used to separate header columns [default: same as `colsep`]

`--hcolend=`*string*
: Override the text placed at the end of each header column [default: same as `colend`]

`--hrowbegin=`*string*
: Override the text placed at the beginning of each header row [default: same as `rowbegin`]

`--hrowsep=`*string*
: Override the text used to separate header rows [default: same as `rowsep`]

`--hrowend=`*string*
: Override the text placed at the end of each header row [default: same as `rowend`]

`--tablebegin=`*string*
: Override the text placed at the beginning of each table [default: “”]

`--tablesep=`*string*
: Override the text used to separate tables [default: “”]

`--tableend=`*string*
: Override the text placed at the end of each table [default: “`\\n`”]

`--quote=`*string*
: Override the text used to begin quoted text [default: “`"`”]

--unquote=string
: Override the text used to end quoted text [default: same as `quote`]

--excel
: Output strings in a format readable by Microsoft Excel

--merge=function
: Specify how to merge data from multiple files [default: "`mean`"]

--showfnames=option
: Add an extra header row showing the filename the data came from [default: "`none`"]

--format=`html`
: Output each table in HTML table format

--noheaders
: Do not output column headers

--colbegin=string
: Override the text placed at the beginning of each data column [default: "`<td>`"]

--colsep=string
: Override the text used to separate data columns [default: " "]

--colend=string
: Override the text placed at the end of each data column [default: "`</td>`"]

--rowbegin=string
: Override the text placed at the beginning of each data row [default: "`<tr>`"]

--rowsep=string
: Override the text used to separate data rows [default: ""]

--rowend=string
: Override the text placed at the end of each data row [default: "`</tr>\\n`"]

--hcolbegin=string
: Override the text placed at the beginning of each header column [default: "`<th>`"]

--hcolsep=string
: Override the text used to separate header columns [default: same as `colsep`]

--hcolend=string
: Override the text placed at the end of each header column [default: "`</th>`"]

`--hrowbegin=`*string*
: Override the text placed at the beginning of each header row [default: same as `rowbegin`]

`--hrowsep=`*string*
: Override the text used to separate header rows [default: same as `rowsep`]

`--hrowend=`*string*
: Override the text placed at the end of each header row [default: same as `rowend`]

`--tablebegin=`*string*
: Override the text placed at the beginning of each table [default: "`<table>\\n`"]

`--tablesep=`*string*
: Override the text used to separate tables [default: ""]

`--tableend=`*string*
: Override the text placed at the end of each table [default: "`</table>\\n`"]

`--quote=`*string*
: Override the text used to begin quoted text [default: ""]

`--unquote=`*string*
: Override the text used to end quoted text [default: same as `quote`]

`--merge=`*function*
: Specify how to merge data from multiple files [default: "`mean`"]

`--showfnames=`*option*
: Add an extra header row showing the filename the data came from [default: "`none`"]

`--format=octave`
: Output each table as an Octave text-format data file

`--noheaders`
: Do not output column headers

`--colbegin=`*string*
: Override the text placed at the beginning of each data column [default: ""]

`--colsep=`*string*
: Override the text used to separate data columns [default: ""]

`--colend=`*string*
: Override the text placed at the end of each data column [default: "`\\n`"]

`--rowbegin=`*string*
: Override the text placed at the beginning of each data row [default: “”]

`--rowend=`*string*
: Override the text placed at the end of each data row [default: “”]

`--hcolbegin=`*string*
: Override the text placed at the beginning of each header column [default: “”]

`--hcolsep=`*string*
: Override the text used to separate header columns [default: “`_`”]

`--hcolend=`*string*
: Override the text placed at the end of each header column [default: “”]

`--hrowbegin=`*string*
: Override the text placed at the beginning of each header row [default: “`#` ”]

`--hrowsep=`*string*
: Override the text used to separate header rows [default: “”]

`--hrowend=`*string*
: Override the text placed at the end of each header row [default: “`\\n`”]

`--tablebegin=`*string*
: Override the text placed at the beginning of each table [default: “”]

`--tablesep=`*string*
: Override the text used to separate tables [default: “`\\n`”]

`--tableend=`*string*
: Override the text placed at the end of each table [default: “”]

`--quote=`*string*
: Override the text used to begin quoted text [default: “”]

`--unquote=`*string*
: Override the text used to end quoted text [default: same as `quote`]

`--merge=`*function*
: Specify how to merge data from multiple files [default: “`mean`”]

`--showfnames=`*option*
: Add an extra header row showing the filename the data came from [default: "`none`"]

`--format=custom`
: Output each table in a completely user-specified format

`--noheaders`
: Do not output column headers

`--colbegin=`*string*
: Specify the text placed at the beginning of each data column [default: ""]

`--colsep=`*string*
: Specify the text used to separate data columns [default: ""]

`--colend=`*string*
: Specify the text placed at the end of each data column [default: ""]

`--rowbegin=`*string*
: Specify the text placed at the beginning of each data row [default: ""]

`--rowsep=`*string*
: Specify the text used to separate data rows [default: ""]

`--rowend=`*string*
: Specify the text placed at the end of each data row [default: ""]

`--hcolbegin=`*string*
: Specify the text placed at the beginning of each header column [default: same as `colbegin`]

`--hcolsep=`*string*
: Specify the text used to separate header columns [default: same as `colsep`]

`--hcolend=`*string*
: Specify the text placed at the end of each header column [default: same as `colend`]

`--hrowbegin=`*string*
: Specify the text placed at the beginning of each header row [default: same as `rowbegin`]

`--hrowsep=`*string*
: Specify the text used to separate header rows [default: same as `rowsep`]

`--hrowend=`*string*
: Specify the text placed at the end of each header row [default: same as `rowend`]

`--tablebegin=`*string*
: Specify the text placed at the beginning of each table [default: ""]

`--tablesep=`*string*
: Specify the text used to separate tables [default: ""]

`--tableend=`*string*
: Specify the text placed at the end of each table [default: ""]

`--quote=`*string*
: Specify the text used to begin quoted text [default: ""]

`--unquote=`*string*
: Specify the text used to end quoted text [default: same as `quote`]

`--excel`
: Output strings in a format readable by Microsoft Excel

`--merge=`*function*
: Specify how to merge data from multiple files [default: "`mean`"]

`--showfnames=`*option*
: Add an extra header row showing the filename the data came from [default: "`none`"]

`--format=latex`
: Output each table as a LaTeX tabular environment

`--dcolumn`
: Use the dcolumn package to align numbers on the decimal point

`--booktabs`
: Use the booktabs package for a more professionally typeset look

`--longtable`
: Use the longtable package to enable multi-page tables

`--merge=`*function*
: Specify how to merge data from multiple files [default: "`mean`"]

`--showfnames=`*option*
: Add an extra header row showing the filename the data came from [default: "`none`"]

`--extract=params`
: Extract the program's run-time parameters and environment variables

`--format=text` [default]
: Output the parameters in plain-text format

  `--include=`*filename*
  : Read from a file the list of keys to output

  `--exclude=`*regexp*
  : Ignore any keys whose name matches a regular expression

  `--sort`
  : Sort the list of parameters alphabetically by key

  `--noenv`
  : Exclude environment variables

  `--noparams`
  : Exclude run-time parameters

  `--envformat=`*template*
  : Format environment variable names using the given template [default: "`%s (environment variable)`"]

  `--columns=`*number*
  : Output the parameters as a 1-, 2-, or 3-column table [default: 1]

  `--colsep=`*string*
  : Override the text used to separate data columns [default: "`: `"]

  `--rowbegin=`*string*
  : Override the text that's output at the start of each data row [default: ""]

  `--rowend=`*string*
  : Override the text that's output at the end of each data row [default: "`\\n`"]

`--format=dumpkeys`
: Output a list of the keys only (i.e., no values)

  `--include=`*filename*
  : Read the list of parameters to output from a given file

  `--exclude=`*regexp*
  : Ignore any keys whose name matches a regular expression

  `--envformat=`*template*
  : Format environment variable names using the given template [default: "`%s (environment variable)`"]

  `--sort`
  : Sort the list of parameters alphabetically by key

  `--noenv`
  : Exclude environment variables

  `--noparams`
  : Exclude run-time parameters

`--format=latex`
: Output the parameters as a LaTeX tabular environment

  `--include=`*filename*
  : Read from a file the list of keys to output

  `--exclude=`*regexp*
  : Ignore any keys whose name matches a regular expression

  `--envformat=`*template*
  : Format environment variable names using the given template [default: "`%s (environment variable)`"]

  `--sort` Sort the list of parameters alphabetically by key

  `--booktabs`
  : Use the booktabs package for a more professionally typeset look

  `--tabularx`
  : Use the tabularx package to enable line wraps within the value column

  `--longtable`
  : Use the longtable package to enable multi-page tables

  `--noenv` Exclude environment variables

  `--noparams`
  : Exclude run-time parameters

`--extract=env`
: Extract the environment in which the program was run

  `--format=sh` [default]
  : Use Bourne shell syntax for setting environment variables

    `--newlines`
    : Separate commands with newlines instead of semicolons

    `--unset` Unset all other environment variables

    `--chdir` Switch to the program's original working directory

  `--format=bash`
  : Use Bourne Again shell syntax for setting environment variables

    `--newlines`
    : Separate commands with newlines instead of semicolons

    `--unset` Unset all other environment variables

    `--chdir` Switch to the program's original working directory

`--format=ksh`
: Use Korn shell syntax for setting environment variables

  `--newlines`
  : Separate commands with newlines instead of semicolons

  `--unset`
  : Unset all other environment variables

  `--chdir`
  : Switch to the program's original working directory

`--format=csh`
: Use C shell syntax for setting environment variables

  `--newlines`
  : Separate commands with newlines instead of semicolons

  `--unset`
  : Unset all other environment variables

  `--chdir`
  : Switch to the program's original working directory

`--format=zsh`
: Use Z shell syntax for setting environment variables

  `--newlines`
  : Separate commands with newlines instead of semicolons

  `--unset`
  : Unset all other environment variables

  `--chdir`
  : Switch to the program's original working directory

`--format=tcsh`
: Use tcsh syntax for setting environment variables

  `--newlines`
  : Separate commands with newlines instead of semicolons

  `--unset`
  : Unset all other environment variables

  `--chdir`
  : Switch to the program's original working directory

`--format=ash`
: Use ash syntax for setting environment variables

  `--newlines`
  : Separate commands with newlines instead of semicolons

  `--unset`
  : Unset all other environment variables

  `--chdir`
  : Switch to the program's original working directory

`--extract=source`
: Extract coNCePTuaL source code

`--format=text` [default]
: Output the source code in plain-text format

  `--indent=`*number*
  : Indent each line by a given number of spaces

  `--wrap=`*number*
  : Wrap the source code into a paragraph with a given character width

The following represent additional clarification for some of the above:

- If `--indent` is specified without an argument, the argument defaults to `2`.
- If `--wrap` is specified without an argument, the argument defaults to `72`.
- The following are examples of the different arguments to the `--columns` option:

`--columns=1` (default)

```
coNCePTuaL version: 1.0
coNCePTuaL backend: c_mpi
Average timer overhead [gettimeofday()]: <1 microsecond
Log creation time: Thu Mar 27 19:22:48 2003
Log completion time: Thu Mar 27 19:22:48 2003
```

`--columns=2`

```
coNCePTuaL version:                        1.0
coNCePTuaL backend:                        c_mpi
Average timer overhead [gettimeofday()]: <1 microsecond
Log creation time:                         Thu Mar 27 19:22:48 2003
Log completion time:                       Thu Mar 27 19:22:48 2003
```

`--columns=3`

```
coNCePTuaL version                      : 1.0
coNCePTuaL backend                      : c_mpi
Average timer overhead [gettimeofday()]: <1 microsecond
Log creation time                       : Thu Mar 27 19:22:48 2003
Log completion time                     : Thu Mar 27 19:22:48 2003
```

- `--dumpkeys` produces suitable input for the `--include` option.
- `--exclude` can be specified repeatedly on the command line.
- `--merge` takes one of `mean` (arithmetic mean), `hmean` (harmonic mean), `min` (minimum), `max` (maximum), `median` (median), or `all` (horizontal concatenation of all data) and applies the function to corresponding data values across all of the input files. `--merge` can also accept a comma-separated list of the above functions, one per data column. This enables a different merge operation to be used for each column. For example, `--merge=min,min,mean` will take the minimum value across all files of each element in the first and second columns and the arithmetic mean across all files of each element in the third column. If the number of comma-separated values differs from the number of columns and `--force-merge` is specified, '`logextract`' will cycle over the given values until all columns are accounted for.
- `--showfnames` prepends to each data table in the input file an extra header line indicating the log file the data was extracted from. This option makes sense only when data is being extracted and primarily when `--merge=all` is specified. `--showfnames`

takes one of `none`, `all`, or `first`. The default is `none`, which doesn't add an extra header row. `all` repeats the filename in each column of the extra header row. `first` outputs the filename in only the first column, leaving the remaining columns with an empty string. The following examples show how a sample data table is formatted with *`--showfnames`* set in turn to each of `none`, `all`, and `first`:

- Set to `none` (the default):

```
"Size","Value"
1,2
2,4
3,6
```

- Set to `all` (filename repeated in each column of the first row):

```
"mydata.log","mydata.log"
"Size","Value"
1,2
2,4
3,6
```

- Set to `first` (filename shown only in the first column of the first row):

```
"mydata.log",""
"Size","Value"
1,2
2,4
3,6
```

- If *`--format`*`=params` is used with both *`--longtable`* and *`--tabularx`*, the generated table will be formatted for use with the `ltxtable` LaTeX package. See `ltxtable`'s documentation for more information.

NOTES

If no filenames are given, '`logextract`' will read from the standard input device. If multiple log files are specified, coNCePTuaL will merge the data values and take all other information from the first file specified. Note, however, that all of the log files must have been produced by the same coNCePTuaL program and that that program must have been run in the same environment. In other words, only the data values may change across log files; everything else must be invariant. See the description of *`--merge`* in the Additional Options section for more information about merging data values from multiple log files.

If the argument provided to any '`logextract`' option begins with an at sign ("`@`"), the value is treated as a filename and is replaced by the file's contents. To specify an non-filename argument that begins with an at sign, merely prepend an additional "`@`":

`--this``=that`
: The option `this` is given the value "`that`".

`--this``=@that`
: The option `this` is set to the contents of the file called '`that`'.

`--this``=@@that`
: The option `this` is given the value "`@that`".

EXAMPLES

For the following examples, we assume that '`results.log`' is the name of a log file produced by a coNCePTuaL program.

Extract the source code that produced '`results.log`':

```
logextract --extract=source results.log
```

Do the same, but indent the code by four spaces then re-wrap it into a 60-column paragraph:

```
logextract --extract=source --indent=4 --wrap=60 results.log
```

Here are a variety of ways to express the same thing:

```
logextract -e source --indent=4 --wrap=60 results.log
logextract -e source --indent=4 results.log --wrap=60
cat results.log | logextract --wrap=60 --indent=4 -e source
```

Output the source code wrapped to 72 columns, with no indentation, and formatted within an HTML preformatted-text block:

```
logextract --extract=source --wrap --before="<PRE>\n" \
  after="</PRE>\n" results.log
```

Extract the data in CSV format and write it to '`results.csv`':

```
logextract --extract=data results.log --output=results.csv
```

Note that `--extract=data` is the default and therefore optional:

```
logextract results.log --output=results.csv
```

'`logextract`' can combine data from multiple log files (using an arithmetic mean by default):

```
logextract results-*.log --output=results.csv
```

Put the data from all of the log files side-by-side and produce a CSV file that Microsoft Excel can read directly:

```
logextract results-*.log --output=results.csv --merge=all \
   --showfnames=first --excel
```

Output '`result.log`''s data in tab-separated-value format:

```
logextract --format=tsv results.log
```

Output the data in space-separated-value format:

```
logextract --colsep=" " results.log
```

Use '`gnuplot`' to draw a PostScript graph of the data:

```
logextract results.log --colsep=" " --noheaders \
  --before=@params.gp | gnuplot > results.eps
```

In the above, the '`params.gp`' file might contain '`gnuplot`' commands such as the following:

```
set terminal postscript eps enhanced color "Times-Roman" 30
set output
set logscale xy
set data style linespoints
set pointsize 3
```

```
plot "-" title "Latency"
```

(There should be an extra blank line at the end of the file because '`logextract`' strips off a trailing newline character whenever it reads a file using "`@`".)

Produce a complete HTML file of the data (noting that `--format=html` produces only tables, not complete documents):

```
logextract --format=html
   --before='<html>\n<head>\n<title>Data</title>\n</head>\n<body>\n' \
   --after='</body>\n</html>\n' results.log
```

Output the data as a LaTeX `tabular`, relying on both the (standard) `dcolumn` and (non-standard) `booktabs` packages for more attractive formatting:

```
logextract --format=latex --dcolumn --booktabs \
  --output=results.tex results.log
```

Output the run-time parameters in the form "*key* `-->` *value*" with all of the arrows aligned:

```
logextract results.log --extract=params --columns=3 --colsep=" --> "
```

Output the run-time parameters as an HTML description list:

```
logextract results.log --extract=params --before='<dl>' \
  --rowbegin='<dt>' --colsep='</dt><dd>' --rowend='</dd>\n' \
  --after='</dl>\n'
```

Restore the exact execution environment that was used to produce '`results.log`', including the current working directory (assuming that '`bash`' is the current command shell):

```
eval `logextract --extract=env --format=bash \
  --unset --chdir results.log`
```

Set all of the environment variables that were used to produce '`results.log`', overwriting–but not removing–whatever environment variables are currently set (assuming that '`tcsh`' is the current command shell):

```
eval `logextract --extract=env --format=tcsh results.log`
```

AUTHOR

Scott Pakin, pakin@lanl.gov

4 Grammar

The coNCePTuaL language was designed to produce precise specifications of network correctness and performance tests yet read like an English-language document that contains a hint of mathematical notation. Unlike more traditional programming languages, coNCePTuaL is more descriptive than imperative. There are no classes, functions, arrays, pointers, or even variable assignments (although expressions can be let-bound to identifiers).[1] The language operates primarily on integers, with support for string constants in a few constructs. A coNCePTuaL program merely describes a communication pattern and the coNCePTuaL compiler generates code to implement that pattern.

As a domain-specific language, coNCePTuaL contains primitives to send and receive messages. It is capable of measuring time, computing statistics, and logging results. It knows that it will be run in a shared-nothing SPMD[2] style with explicit message-passing. As a result of its special-purpose design coNCePTuaL can express communication patterns in a clearer and terser style than is possible using a general-purpose programming language.

The coNCePTuaL language is case-insensitive. `Hello` is the same as `HELLO` or `hello`. Furthermore, whitespace is insignificant; one space has the same meaning as multiple spaces. Comments are designated with a '`#`' character and extend to the end of the line.

We now describe the coNCePTuaL grammar in a bottom-up manner, i.e., starting from primitives and working up to complete programs. Note that many of the sections in this chapter use the following syntax to formally describe language elements:

⟨*nonterminal*⟩	a placeholder for a list of language primitives and additional placeholders
::=	"is defined as"
`KEYWORD`	a primitive with special meaning to the language
[. . .]	optional items
(. . .)	grouping of multiple items into one
*	zero or more occurrences of the preceding item
+	one or more occurrences of the preceding item
|	either the item to the left or the item to the right but not both

4.1 Primitives

At the lowest level, coNCePTuaL programs are composed of identifiers, strings, and integers (and a modicum of punctuation). Identifiers consist of a letter followed by zero or more alphanumerics or underscores. '`potato`', '`x`', and '`This_is_program_123`' are all examples of valid identifiers. Identifiers are used for two purposes: variables and keywords.

[1] coNCePTuaL is not even Turing-complete. That is, it cannot perform arbitrary computations.

[2] Single Program, Multiple Data

Variables—referred to in the formal grammar as ⟨*ident*⟩s—can be bound but not assigned. That is, once a variable is given a value it retains that value for the entire scope although it may be given a different value within a subordinate scope. All variables are of integer type. There are a number of variables that are predeclared and maintained automatically by coNCePTuaL. These are listed and described in Section A.2 [Predeclared variables], page 127. Predeclared variables can be used by coNCePTuaL programs but cannot be redeclared; an attempt to do so will result in a compile-time error message.

Keywords introduce actions. For example, `SEND` and `RECEIVE` are keywords. (A complete list of coNCePTuaL keywords is presented in Section A.1 [Keywords], page 123.) Most keywords can appear in multiple forms. For example, `OUTPUT` and `OUTPUTS` are synonymous, as are `COMPUTE` and `COMPUTES`, `A` and `AN`, and `TASK` and `TASKS`. The intention is for programs to use whichever sounds better in an English-language sentence. Keywords may not be used as variable names; an attempt to do so will cause the compiler to output a parse error.

Although identifiers are case insensitive—`SEND` is the same as '`send`' is the same as '`sENd`'—to increase clarity, this manual presents keywords in uppercase and variables in lowercase.

Strings consist of double-quoted text. Within a string—and only within a string—whitespace and case are significant. Use '`\"`' for a double-quote character, '`\\`' for a backslash, and '`\n`' for a newline character. For example, the string `"October 2004"` represents the text "October 2004" and `"I store \"stuff\" in C:\\MyStuff."` represents "I store "stuff" in C:\MyStuff." Within the double quotes verbatim newline characters and all subsequent spaces are replaced with a single space, as in the following example:

```
"This string
originally contained
     some newline characters."
```

⇒ "This string originally contained some newline characters."

Integers consist of an optional '`+`' or '`-`' followed by one or more digits followed by an optional multiplier. This multiplier is unique to coNCePTuaL and consists of one of the following four letters:

'`K`' (kilo) multiplies the integer by 1,024

'`M`' (mega) multiplies the integer by 1,048,576

'`G`' (giga) multiplies the integer by 1,073,741,824

'`T`' (tera) multiplies the integer by 1,099,511,627,776

In addition, a multiplier can be '`E`' (exponent) followed by a positive integer. An '`E`' multiplier multiplies the base integer by 10 raised to the power of the alternate integer.

Some examples of valid integers include '`2004`', '`-42`', '`64K`' (= 65,536), and '`8E3`' (= 8,000).

4.2 Expressions

Expressions, as in any language, are a combination of primitives and other expressions in a semantically meaningful juxtaposition. coNCePTuaL provides arithmetic expressions, which evaluate to a number, and relational expressions, which evaluate to either TRUE or

FALSE. In addition, coNCePTuaL provides the notion of an *aggregate expression*, which represents a function (e.g., statistical mean) applied to every value taken on by an arithmetic expression during the run of a program.

4.2.1 Arithmetic expressions

coNCePTuaL supports a variety of arithmetic expressions. The following is the language's order of operations from highest to lowest precedence:

unary	'+', '-', NOT, '⟨*function*⟩(⟨*expr*⟩, ...)', 'REAL(⟨*expr*⟩)'
power	'**'
multiplicative	'*', '/', MOD, '<<', '>>', AND
additive	'+', '-', OR, XOR
conditional	'⟨*expr*⟩ IF ⟨*rel_expr*⟩ OTHERWISE ⟨*expr*⟩'

In addition, as in most programming languages, parentheses can be used to group subexpressions.

AND, OR, XOR, and NOT are bitwise operators. Hence, for example, '3 OR 5' is equal to '7'.

'<<' and '>>' are bit-shift operators. That is, 'a << b' is the coNCePTuaL equivalent of the mathematical expression $a \times 2^b$ and 'a >> b' is the coNCePTuaL equivalent of the mathematical expression $a \div 2^b$.

MOD is a modulo (i.e., remainder) operator: '10 MOD 3' returns '1'. 'MOD' is guaranteed to return a nonnegative remainder. Hence, '16 MOD 7' and '16 MOD -7' both return '2' even though '-5' is also mathematically valid. Similarly, '-16 MOD 7' and '-16 MOD -7' both return '5' even though '-2' is also mathematically valid.

The function calls allowed in '⟨*function*⟩(⟨*expr*⟩, ...)' are listed and described in Section 4.2.2 [Built-in functions], page 49. All functions take one or more arithmetic expressions as an argument. The operator '*' represents multiplication; '/' represents division; and '**' represents exponentiation (i.e., 'x ** y' $\equiv \lfloor x^y \rfloor$). Note that 0^y generates a run-time error for $y \leq 0$.

A conditional expression '⟨*expr1*⟩ IF ⟨*rel_expr*⟩ OTHERWISE ⟨*expr2*⟩' evaluates to ⟨*expr1*⟩ if the relational expression ⟨*rel_expr*⟩ evaluates to TRUE and ⟨*expr2*⟩ if ⟨*rel_expr*⟩ evaluates to FALSE.[3] Relational expressions are described in Section 4.2.5 [Relational expressions], page 60. As some examples of conditional expressions, '666 IF 2+2=5 OTHERWISE 777' returns '777' while '666 IF 2+2=4 OTHERWISE 777' returns '666'.

All operations proceed left-to-right except power and conditional expressions, which proceed right-to-left. That is, '4-3-2' means $(4-3)-2$ but '4**3**2' means $4^{(3^2)}$. Similarly, '2 IF p=0 OTHERWISE 1 IF p=1 OTHERWISE 0' associates like '2 IF p=0 OTHERWISE (1 IF p=1 OTHERWISE 0)', not like '(2 IF p=0 OTHERWISE 1) IF p=1 OTHERWISE 0'.

[3] It is therefore analogous to '⟨*rel_expr*⟩ ? ⟨*expr1*⟩ : ⟨*expr2*⟩' in the C programming language.

Evaluation contexts

coNCePTuaL normally evaluates arithmetic expressions in "integer context", meaning that each subexpression is truncated to the nearest integer after being evaluated. Hence, '`24/5*5`' is '`20`', not '`24`', because '`24/5*5`' $= \lfloor 24 \div 5 \rfloor \times 5 = 4 \times 5 = 20$. There are a few situations, however, in which coNCePTuaL evaluates expressions in "floating-point context", meaning that no truncation occurs:

- within an `OUTPUTS` statement (see Section 4.5.2 [Writing to standard output], page 71)
- within a `LOGS` statement (see Section 4.5.3 [Writing to a log file], page 71)
- within a `BACKEND EXECUTES` statement (see Section 4.6.6 [Injecting arbitrary code], page 77)
- within a range in a `FOR EACH` statement (see Section 4.7.2 [Iterating], page 79) when coNCePTuaL is unable to find an arithmetic or geometric progression by evaluating the component ⟨*expr*⟩s in integer context

Within any of the preceding statements, the expression '`24/5*5`' evaluates to 24. Furthermore, the expression '`24/5`' evaluates to 4.8, which is a number that can't be entered directly in a coNCePTuaL program. (The language supports only integral constants, as mentioned in Section 4.1 [Primitives], page 46.)

The coNCePTuaL language provides a special form called `REAL` which resembles a single-argument function. When evaluated in floating-point context, `REAL` returns its argument evaluated normally, as if '`REAL`' were absent. When evaluated in integer context, however, `REAL` evalutes its argument in floating-point context and then rounds the result to the nearest integer. As an example, '`9/2 + 1/2`' is '`4`' in integer context because '`9/2 + 1/2`' $= \lfloor 9/2 \rfloor + \lfloor 1/2 \rfloor = 4 + 0 = 4$. However, '`REAL(9/2 + 1/2)`' is '`5`' in integer context because '`REAL(9/2 + 1/2)`' $= \lfloor 9/2 + 1/2 + 0.5 \rfloor = \lfloor 5 + 0.5 \rfloor = 5$.

4.2.2 Built-in functions

In addition to the operators described in Section 4.2.1 [Arithmetic expressions], page 48, coNCePTuaL contains a number of built-in functions that perform a variety of arithmetic operations that are often found to be useful in network correctness and performance testing codes. These include simple functions that map one number to another as well as a set of topology-specific functions that help implement communication across various topologies, specifically n-ary trees, meshes, tori, and k-nomial trees. coNCePTuaL currently supports the following functions:

- `ABS`
- `BITS`
- `CBRT`
- `FACTOR10`
- `LOG10`
- `MAX`
- `MIN`
- `ROOT`

- SQRT
- CEILING
- FLOOR
- ROUND
- TREE_PARENT
- TREE_CHILD
- KNOMIAL_PARENT
- KNOMIAL_CHILD
- KNOMIAL_CHILDREN
- MESH_NEIGHBOR
- MESH_COORDINATE
- TORUS_NEIGHBOR
- TORUS_COORDINATE
- RANDOM_UNIFORM
- RANDOM_GAUSSIAN
- RANDOM_POISSON

All of the above take as an argument one or more integers (which may be the result of an arithmetic expression). The following sections describe each function in turn.

Integer functions

ABS returns the absolute value of its argument. For example, 'ABS(99)' and 'ABS(-99)' are both '99'.

BITS returns the minimum number of bits needed to store its argument. For example, 'BITS(12345)' is '14' because 2^{14} is 16,384, which is larger than 12,345, while 2^{13} is 8,192, which is too small. 'BITS(0)' is defined to be '0'. Essentially, 'BITS(x)' represents $\lceil \log_2 x \rceil$, i.e., the ceiling of the base-2 logarithm of x. Negative numbers are treated as their two's-complement equivalent. For example, 'BITS(-1)' returns '32' on a 32-bit system and '64' on a 64-bit system.

CBRT is an integer cube root function. It is essentially just syntactic sugar for the more general ROOT function: 'CBRT(x)' $\equiv$ 'ROOT(3, x)'.

FACTOR10 rounds its argument down (more precisely, towards zero) to the largest single-digit factor of an integral power of 10. 'FACTOR10(4975)' is therefore '4000'. Similarly, 'FACTOR10(-4975)' is '-4000'.

'LOG10(x)' is $\lfloor \log x \rfloor$, i.e., the floor of the base-10 logarithm of x. For instance, 'LOG10(12345)' is '4' because 10^4 is the largest integral power of 10 that does not exceed 12,345.

`MIN` and `MAX` return, respectively, the minimum and maximum value in a list of numbers. Unlike the other built-in functions, `MIN` and `MAX` accept an arbitrary number of arguments (but at least one). For example, '`MIN(8,6,7,5,3,0,9)`' is '`0`' and '`MAX(8,6,7,5,3,0,9)`' is '`9`'.

'`ROOT(`n`, `x`)`' returns $\sqrt[n]{x}$, i.e., the nth root of x. More precisely, it returns the largest integer r such that $r^n \leq x$. `ROOT` is not currently defined on negative values of x but this may change in a future release of coNCePTuaL. As an example of `ROOT` usage, '`ROOT(5, 245)`' is '`3`' because $3^5 = 243 \leq 245$ but $4^5 = 1024 > 245$. Similarly, '`ROOT(2, 16)`' = '`4`'; '`ROOT(3, 27)`' = '`3`'; '`ROOT(0, 0)`' and '`ROOT(4, -245)`' each return a run-time error; and, '`ROOT(-3, 8)`' = '`0`' (because '`ROOT(-3, 8)`' = '`1/ROOT(3, 8)`' = '`1/2`' = '`0`').

`SQRT` is an integer square root function. It is essentially just syntactic sugar for the more general `ROOT` function: '`SQRT(`x`)`' ≡ '`ROOT(2, `x`)`'.

Floating-point functions

As stated in Section 4.2.1 [Arithmetic expressions], page 48, there are certain constructs in which expressions are evaluated in floating-point context instead of integer context. In such constructs, all of coNCePTuaL's built-in functions return floating-point values. Furthermore, the `CBRT`, `LOG10`, `ROOT`, and `SQRT` functions compute floating-point results, not integer results which are coerced into floating-point format.

The following functions are not meaningful in integer context but are in floating-point context:

- `CEILING`
- `FLOOR`
- `ROUND`

`CEILING` returns the smallest integer not less than its argument. For example, '`CEILING(-7777/10)`' is '`-777`'. (-778 is less than -777.7 while -777 is not less than -777.7.)

`FLOOR` returns the largest integer not greater than its argument. For example, '`FLOOR(-7777/10)`' is '`-778`'. (-778 is not greater than -777.7 while -777 is greater than -777.7.)

`ROUND` rounds its argument to the nearest integer. For example, '`ROUND(-7777/10)`' is '`-778`'.

It is not an error to use `CEILING`, `FLOOR`, and `ROUND` in an integer context; each function merely return its argument unmodified.

n-ary tree functions

n-ary trees are used quite frequently in communication patterns because they require only logarithmic time (in the number of tasks) for a message to propagate from the root to a leaf. coNCePTuaL supports n-ary trees in the form of the `TREE_PARENT` and `TREE_CHILD` functions.

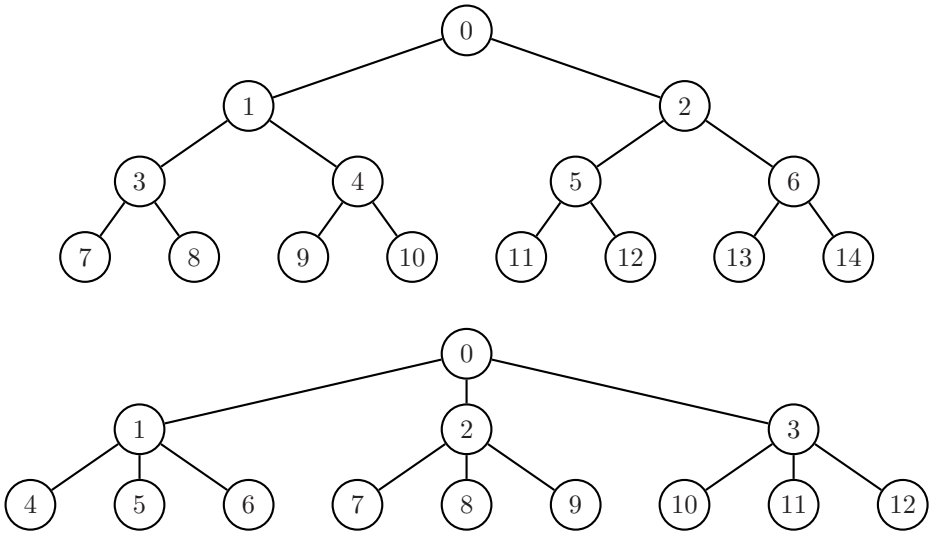
TREE_PARENT (*task_ID* [, *fan-out*]) Function

`TREE_PARENT` takes a task number and an optional tree fan-out (n) and returns the task's parent in an n-ary tree. n defaults to '`2`', i.e., a binary tree. Taking the `TREE_PARENT` of any task less than 1 returns the value '`-1`'.

TREE_CHILD (*task_ID, child* [, *fan-out*]) Function

TREE_CHILD takes a task number, a child number ($0 \leq i < N$), and an optional tree fan-out (n), which again defaults to '2'. It returns the task number corresponding to the given task's *child*th child.

The following illustrations show how tasks are numbered in, respectively, a 2-ary and a 3-ary tree:



As shown by the 2-ary tree, task 1's children are task 3 and task 4. Therefore, 'TREE_PARENT(3)' and 'TREE_PARENT(4)' are both '1'; 'TREE_CHILD(1, 0)' is '3'; and, 'TREE_CHILD(1, 1)' is '4'. In a 3-ary tree, each task has three children. Hence, the following expressions hold:

- 'TREE_PARENT(7, 3)' ⇒ '2'
- 'TREE_PARENT(8, 3)' ⇒ '2'
- 'TREE_PARENT(9, 3)' ⇒ '2'
- 'TREE_CHILD(2, 0, 3)' ⇒ '7'
- 'TREE_CHILD(2, 1, 3)' ⇒ '8'
- 'TREE_CHILD(2, 2, 3)' ⇒ '9'

k-nomial tree functions

k-nomial trees are an efficient way to implement collective-communication operations in software. Unlike in an *n*-ary tree, the number of children in a *k*-nomial tree decreases with increasing task depth (i.e., no task has more children than the root). The advantage is that the tasks that start communicating earlier perform more work, which reduces the total latency of the collective operation. In contrast, in an *n*-ary tree, the tasks that start communicating earlier finish earlier, at the expense of increased total latency. coNCePTuaL supports *k*-nomial trees via the KNOMIAL_PARENT, KNOMIAL_CHILDREN, and KNOMIAL_CHILD functions, as described below.

KNOMIAL_PARENT (*task_ID* [, *fan_out* [, *num_tasks*]]) Function

`KNOMIAL_PARENT` takes a task number, the tree fan-out factor (the "*k*" in "*k*-ary"), and the number of tasks in the tree. It returns the task ID of the given task's parent. *fan_out* defaults to '`2`' and the number of tasks defaults to `num_tasks` (see Section A.2 [Predeclared variables], page 127).

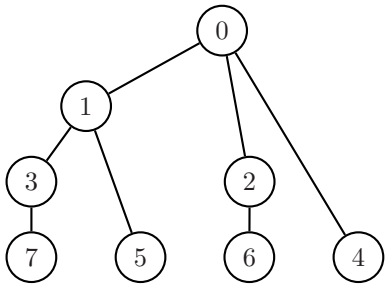
KNOMIAL_CHILDREN (*task_ID* [, *fan_out* [, *num_tasks*]]) Function

`KNOMIAL_CHILDREN` takes the same arguments as `KNOMIAL_PARENT` but returns the number of immediate descendents the given task has.

KNOMIAL_CHILD (*task_ID*, *child* [, *fan_out* [, *num_tasks*]]) Function

`KNOMIAL_CHILD` takes a task number, a child number ($0 \leq i <$ '`KNOMIAL_CHILDREN(...)`'), the tree fan-out factor, and the number of tasks in the tree. It returns the task number corresponding to the given task's ith child. As in `KNOMIAL_PARENT` and `KNOMIAL_CHILDREN`, *fan_out* defaults to '`2`' and the number of tasks defaults to `num_tasks` (see Section A.2 [Predeclared variables], page 127).

The following figure shows how coNCePTuaL numbers tasks in a k-nomial tree with $k = 2$ (a.k.a. a 2-nomial or binomial tree).

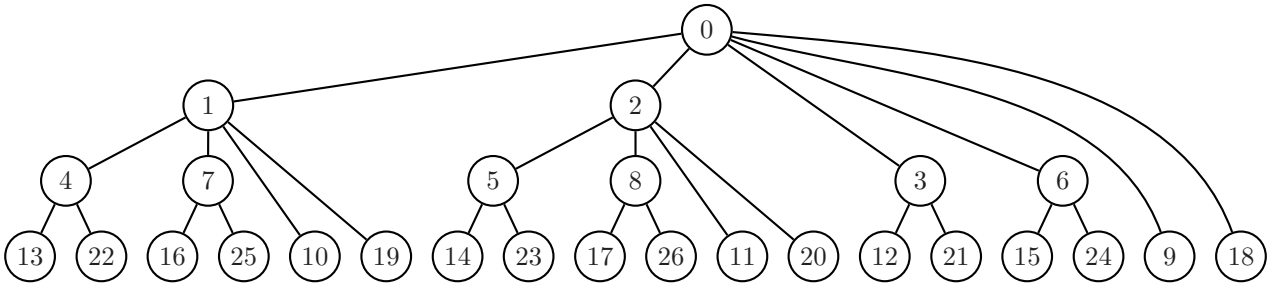


The figure is structured with time flowing downwards. That is, for a multicast operation expressed over a 2-nomial tree, task 0 sends a message to task 1 in the first time step. Then, task 0 sends to task 2 while task 1 sends to task 3. In the final step, task 0 sends to task 4, task 1 sends to task 5, task 2 sends to task 6, and task 3 sends to task 7—all concurrently. The following expressions also hold, assuming there are a total of eight tasks in the computation:

- '`KNOMIAL_PARENT(0)`' ⇒ '`-1`'
- '`KNOMIAL_PARENT(1)`' ⇒ '`0`'
- '`KNOMIAL_CHILDREN(1)`' ⇒ '`2`'
- '`KNOMIAL_CHILD(1, 0)`' ⇒ '`3`'
- '`KNOMIAL_CHILD(1, 1)`' ⇒ '`5`'
- '`KNOMIAL_CHILDREN(7)`' ⇒ '`0`'
- '`KNOMIAL_CHILD(7, 0)`' ⇒ '`-1`'

k-nomial trees for $k > 2$ are much less common in practice than 2-nomial trees. However, they may perform well when a task has sufficient bandwidth to support multiple,

simultaneous, outgoing messages. For example, a trinomial tree (i.e., a k-nomial tree with $k = 3$) should exhibit good performance if there is enough bandwidth to send two messages simultaneously. The following illustration shows how coNCePTuaL constructs a 27-task trinomial tree:



As before, time flows downward (assuming a multicast operation) and tasks are expected to communicate with their children in order. The following are some coNCePTuaL k-nomial tree expressions and their evaluations, assuming `num_tasks` is '27':

- '`KNOMIAL_PARENT(0, 3)`' ⇒ '`-1`'
- '`KNOMIAL_PARENT(2, 3)`' ⇒ '`0`'
- '`KNOMIAL_CHILDREN(2, 3)`' ⇒ '`4`'
- '`KNOMIAL_CHILD(2, 0, 3)`' ⇒ '`5`'
- '`KNOMIAL_CHILD(2, 1, 3)`' ⇒ '`8`'
- '`KNOMIAL_CHILD(2, 2, 3)`' ⇒ '`11`'
- '`KNOMIAL_CHILD(2, 3, 3)`' ⇒ '`20`'
- '`KNOMIAL_CHILD(2, 4, 3)`' ⇒ '`-1`'
- '`KNOMIAL_CHILDREN(8, 3)`' ⇒ '`2`'
- '`KNOMIAL_CHILDREN(8, 3, 26)`' ⇒ '`1`'
- '`KNOMIAL_CHILDREN(8, 3, 10)`' ⇒ '`0`'

Mesh functions

coNCePTuaL provides two functions, `MESH_NEIGHBOR` and `MESH_COORDINATE`, that help treat (linear) task IDs as positions on a multidimensional mesh. Each of these functions takes a variable number of arguments, determined by the dimensionality of the mesh (1-D, 2-D, or 3-D).

MESH_NEIGHBOR (*task_ID, width, x_offset* [*, height, y_offset* [*, depth, z_offset*]]) Function

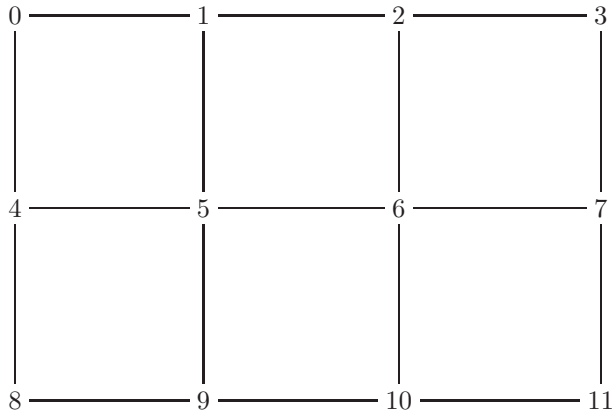
`MESH_NEIGHBOR` returns a task's neighbor on a 1-D, 2-D, or 3-D mesh. It always takes a task number, the mesh's width, and the desired x offset from the given task. For a 2-D or 3-D mesh, the next two arguments are the mesh's height and the desired y offset from the given task. For a 3-D mesh only, the next two arguments are the mesh's depth and the desired z offset from the given task. Offsets that move off the mesh cause `MESH_NEIGHBOR` to return the value '`-1`'.

MESH_COORDINATE (*task_ID, coordinate, width* [, *height* [, *depth*]]) Function

MESH_COORDINATE returns a task's *x*, *y*, or *z* coordinate on a 1-D, 2-D, or 3-D mesh. The first argument to MESH_COORDINATE is a task number. The second argument should be '0' to calculate an *x* coordinate, '1' to calculate a *y* coordinate, or '2' to calculate a *z* coordinate. The remaining arguments are the mesh's width, height, and depth, respectively. The height can be omitted for a 1-D mesh and the depth can be omitted for a 2-D mesh. (Both default to '1'.)

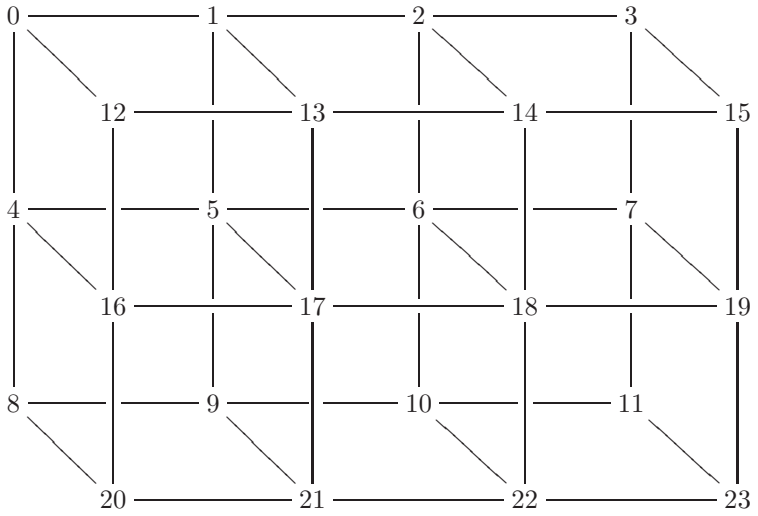
MESH_NEIGHBOR and MESH_COORDINATE number tasks following the right-hand rule: left-to-right, then top-to-bottom, and finally back-to-front, as shown in the following illustrations of a 4-element (1-D) mesh, a 4×3 (2-D) mesh, and a $4 \times 3 \times 2$ (3-D) mesh. Examples of MESH_NEIGHBOR and MESH_COORDINATE for 1-D, 2-D, and 3-D meshes follow the corresponding illustration.

- 'MESH_NEIGHBOR(0, 4, -1)' ⇒ '-1'
- 'MESH_NEIGHBOR(0, 4, +1)' ⇒ '1'
- 'MESH_NEIGHBOR(1, 4, -1)' ⇒ '0'
- 'MESH_NEIGHBOR(1, 4, +1)' ⇒ '2'
- 'MESH_NEIGHBOR(2, 4, -1)' ⇒ '1'
- 'MESH_NEIGHBOR(2, 4, +1)' ⇒ '3'
- 'MESH_NEIGHBOR(3, 4, -1)' ⇒ '2'
- 'MESH_NEIGHBOR(3, 4, +1)' ⇒ '-1'
- 'MESH_COORDINATE(-1, 0, 4)' ⇒ '-1'
- 'MESH_COORDINATE(0, 0, 4)' ⇒ '0'
- 'MESH_COORDINATE(1, 0, 4)' ⇒ '1'
- 'MESH_COORDINATE(2, 0, 4)' ⇒ '2'
- 'MESH_COORDINATE(3, 0, 4)' ⇒ '3'
- 'MESH_COORDINATE(4, 0, 4)' ⇒ '-1'
- 'MESH_COORDINATE(2, 1, 4)' ⇒ '0'
- 'MESH_COORDINATE(2, 2, 4)' ⇒ '0'



- ‘MESH_NEIGHBOR(5, 4, -1, 3, -1)’ ⇒ ‘0’
- ‘MESH_NEIGHBOR(5, 4, 0, 3, -1)’ ⇒ ‘1’
- ‘MESH_NEIGHBOR(5, 4, +1, 3, -1)’ ⇒ ‘2’
- ‘MESH_NEIGHBOR(5, 4, -1, 3, 0)’ ⇒ ‘4’
- ‘MESH_NEIGHBOR(5, 4, 0, 3, 0)’ ⇒ ‘5’
- ‘MESH_NEIGHBOR(5, 4, +1, 3, 0)’ ⇒ ‘6’
- ‘MESH_NEIGHBOR(5, 4, -1, 3, +1)’ ⇒ ‘8’
- ‘MESH_NEIGHBOR(5, 4, 0, 3, +1)’ ⇒ ‘9’
- ‘MESH_NEIGHBOR(5, 4, +1, 3, +1)’ ⇒ ‘10’

- ‘MESH_COORDINATE(1, 0, 4, 3)’ ⇒ ‘1’
- ‘MESH_COORDINATE(6, 0, 4, 3)’ ⇒ ‘2’
- ‘MESH_COORDINATE(6, 1, 4, 3)’ ⇒ ‘1’
- ‘MESH_COORDINATE(6, 2, 4, 3)’ ⇒ ‘0’
- ‘MESH_COORDINATE(8, 0, 4, 3)’ ⇒ ‘0’
- ‘MESH_COORDINATE(8, 1, 4, 3)’ ⇒ ‘2’
- ‘MESH_COORDINATE(12, 0, 4, 3)’ ⇒ ‘-1’



- 'MESH_NEIGHBOR(0, 4, 0, 3, 0, 2, +1)' ⇒ '12'
- 'MESH_NEIGHBOR(0, 4, 0, 3, +1, 2, 0)' ⇒ '4'
- 'MESH_NEIGHBOR(0, 4, +1, 3, 0, 2, 0)' ⇒ '1'
- 'MESH_NEIGHBOR(0, 4, +1, 3, +1, 2, +1)' ⇒ '17'
- 'MESH_NEIGHBOR(17, 4, +2, 3, -1, 2, -1)' ⇒ '3'
- 'MESH_NEIGHBOR(23, 4, +1, 3, +1, 2, +1)' ⇒ '-1'

- 'MESH_COORDINATE(-5, 0, 4, 3, 2)' ⇒ '-1'
- 'MESH_COORDINATE(1, 0, 4, 3, 2)' ⇒ '1'
- 'MESH_COORDINATE(6, 0, 4, 3, 2)' ⇒ '2'
- 'MESH_COORDINATE(6, 1, 4, 3, 2)' ⇒ '1'
- 'MESH_COORDINATE(6, 2, 4, 3, 2)' ⇒ '0'
- 'MESH_COORDINATE(18, 0, 4, 3, 2)' ⇒ '2'
- 'MESH_COORDINATE(18, 1, 4, 3, 2)' ⇒ '1'
- 'MESH_COORDINATE(18, 2, 4, 3, 2)' ⇒ '1'
- 'MESH_COORDINATE(18, 3, 4, 3, 2)' ⇒ error Invalid coordinate

Torus functions

A torus is a mesh with wraparound edges. coNCePTuaL provides analogues to MESH_NEIGHBOR and MESH_COORDINATE called TORUS_NEIGHBOR and TORUS_COORDINATE which enable (linear) task IDs to be treated as positions on a 1-D, 2-D, or 3-D torus.

TORUS_NEIGHBOR (*task_ID, width, x_offset* [, *height, y_offset* [, *depth, z_offset*]]) Function

TORUS_NEIGHBOR takes the same arguments as MESH_NEIGHBOR but calculates neighbors on a torus instead of a mesh. See the description of MESH_NEIGHBOR for explanations of the arguments. While task offsets that go out-of-bounds on a mesh return '-1', such offsets merely wrap around a torus.

TORUS_COORDINATE (*task_ID, coordinate, width* [, *height* [, *depth*]]) Function

TORUS_COORDINATE returns a task's *x*, *y*, or *z* coordinate on a 1-D, 2-D, or 3-D torus. It performs exactly the same function as MESH_COORDINATE and is aliased in the language merely for symmetry. See the description of TORUS_NEIGHBOR for details.

- 'TORUS_NEIGHBOR(0, 4, -1)' ⇒ '3'
- 'TORUS_NEIGHBOR(0, 4, +1)' ⇒ '1'
- 'TORUS_NEIGHBOR(1, 4, -1)' ⇒ '0'
- 'TORUS_NEIGHBOR(1, 4, +1)' ⇒ '2'
- 'TORUS_NEIGHBOR(2, 4, -1)' ⇒ '1'
- 'TORUS_NEIGHBOR(2, 4, +1)' ⇒ '3'
- 'TORUS_NEIGHBOR(3, 4, -1)' ⇒ '2'

- ‘TORUS_NEIGHBOR(3, 4, +1)’ ⇒ ‘0’

- ‘TORUS_COORDINATE(-1, 0, 4)’ ⇒ ‘-1’
- ‘TORUS_COORDINATE(0, 0, 4)’ ⇒ ‘0’
- ‘TORUS_COORDINATE(1, 0, 4)’ ⇒ ‘1’
- ‘TORUS_COORDINATE(2, 0, 4)’ ⇒ ‘2’
- ‘TORUS_COORDINATE(3, 0, 4)’ ⇒ ‘3’
- ‘TORUS_COORDINATE(4, 0, 4)’ ⇒ ‘-1’
- ‘TORUS_COORDINATE(2, 1, 4)’ ⇒ ‘0’
- ‘TORUS_COORDINATE(2, 2, 4)’ ⇒ ‘0’

- ‘TORUS_NEIGHBOR(0, 4, -1, 3, -1)’ ⇒ ‘11’
- ‘TORUS_NEIGHBOR(0, 4, 0, 3, -1)’ ⇒ ‘8’
- ‘TORUS_NEIGHBOR(0, 4, +1, 3, -1)’ ⇒ ‘9’
- ‘TORUS_NEIGHBOR(0, 4, -1, 3, 0)’ ⇒ ‘3’
- ‘TORUS_NEIGHBOR(0, 4, 0, 3, 0)’ ⇒ ‘0’
- ‘TORUS_NEIGHBOR(0, 4, +1, 3, 0)’ ⇒ ‘1’
- ‘TORUS_NEIGHBOR(0, 4, -1, 3, +1)’ ⇒ ‘7’
- ‘TORUS_NEIGHBOR(0, 4, 0, 3, +1)’ ⇒ ‘4’
- ‘TORUS_NEIGHBOR(0, 4, +1, 3, +1)’ ⇒ ‘5’

- ‘TORUS_NEIGHBOR(23, 4, +1, 3, +1, 2, +1)’ ⇒ ‘0’
- ‘TORUS_NEIGHBOR(23, 4, +2, 3, +2, 2, +2)’ ⇒ ‘17’
- ‘TORUS_NEIGHBOR(23, 4, +3, 3, +3, 2, +3)’ ⇒ ‘10’

- ‘TORUS_COORDINATE(1, 0, 4, 3)’ ⇒ ‘1’
- ‘TORUS_COORDINATE(6, 0, 4, 3)’ ⇒ ‘2’
- ‘TORUS_COORDINATE(6, 1, 4, 3)’ ⇒ ‘1’
- ‘TORUS_COORDINATE(6, 2, 4, 3)’ ⇒ ‘0’
- ‘TORUS_COORDINATE(8, 0, 4, 3)’ ⇒ ‘0’
- ‘TORUS_COORDINATE(8, 1, 4, 3)’ ⇒ ‘2’
- ‘TORUS_COORDINATE(12, 0, 4, 3)’ ⇒ ‘-1’

- ‘TORUS_COORDINATE(-5, 0, 4, 3, 2)’ ⇒ ‘-1’
- ‘TORUS_COORDINATE(1, 0, 4, 3, 2)’ ⇒ ‘1’
- ‘TORUS_COORDINATE(6, 0, 4, 3, 2)’ ⇒ ‘2’
- ‘TORUS_COORDINATE(6, 1, 4, 3, 2)’ ⇒ ‘1’
- ‘TORUS_COORDINATE(6, 2, 4, 3, 2)’ ⇒ ‘0’
- ‘TORUS_COORDINATE(18, 0, 4, 3, 2)’ ⇒ ‘2’
- ‘TORUS_COORDINATE(18, 1, 4, 3, 2)’ ⇒ ‘1’
- ‘TORUS_COORDINATE(18, 2, 4, 3, 2)’ ⇒ ‘1’
- ‘TORUS_COORDINATE(18, 3, 4, 3, 2)’ ⇒ error Invalid coordinate

Random-number functions

coNCePTuaL programs can utilize randomness in one of two ways. The functions described below are *unsynchronized* across tasks. That is, they can—and usually do—return a different value to each task on each invocation. One consequence is that these functions are not permitted within a task expression (see Section 4.3 [Task descriptions], page 61) because randomness would cause the tasks to disagree about who the sources and targets of an operation are. In contrast, the `A RANDOM TASK` construct described in Section 4.7.3 [Binding variables], page 83 returns a value guaranteed to be synchronized across tasks and thereby enables random-task selection.

RANDOM_UNIFORM (*lower_bound*, *upper_bound*) Function
Return a number selected at random from a uniform distribution over the range [*lower_bound*, *upper_bound*).

RANDOM_GAUSSIAN (*mean*, *stddev*) Function
Return a number selected at random from a Gaussian distribution with mean *mean* and standard deviation *stddev*.

RANDOM_POISSON (*mean*) Function
Return an integer selected at random from a Poisson distribution with mean *mean* and standard deviation $\sqrt{mean}$.

4.2.3 Aggregate expressions

Aggregate expressions (⟨*aggr_expr*⟩s) are currently used exclusively by the `LOGS` statement. They represent an expression with a given function applied to the aggregate of all (dynamic) instances of that expression. ⟨*aggr_expr*⟩s take one of four forms:

1. `EACH` ⟨*expr*⟩
   (Note: `EACH` can be omitted.)
2. `THE` ⟨*expr*⟩
3. `THE` ⟨*aggr_func*⟩ `OF THE` ⟨*expr*⟩
   (Note: `OF THE` can be shortened to just `OF` or omitted altogether.)
4. `A HISTOGRAM OF THE` ⟨*expr*⟩
   (Note: The `THE` can be omitted.)

(In the above, ⟨*expr*⟩ refers to an arithmetic expression defined in Section 4.2.1 [Arithmetic expressions], page 48 and ⟨*aggr_func*⟩ refers to one of the functions defined in Section 4.2.4 [Aggregate functions], page 60.)

The first form does not summarize ⟨*expr*⟩; every individual instance of ⟨*expr*⟩ is utilized. The second form asserts that ⟨*expr*⟩ is a constant (i.e., all values are identical) and utilizes that constant.[4] The third form applies ⟨*aggr_func*⟩ to the set of all values of ⟨*expr*⟩ and utilizes the result of that function. The fourth form produces a histogram of all values of ⟨*expr*⟩, i.e., a list of `{`*unique value*, *tally*`}` pairs, sorted by *unique value*.

[4] The program aborts with a run-time error if ⟨*expr*⟩ is not a constant.

4.2.4 Aggregate functions

The following functions, referred to collectively as ⟨*aggr_func*⟩s, may be used in an aggregate expression (see Section 4.2.3 [Aggregate expressions], page 59).

- `ARITHMETIC MEAN`
- `HARMONIC MEAN`
- `GEOMETRIC MEAN`
- `MEDIAN`
- `STANDARD DEVIATION`
- `VARIANCE`
- `SUM`
- `MINIMUM`
- `MAXIMUM`
- `FINAL`

`ARITHMETIC MEAN` can be abbreviated to simply `MEAN`. `MEDIAN` is the value such that there are as many larger as smaller values. If there are an even number of values, `MEDIAN` is the arithmetic mean of the two medians. `FINAL` returns only the final value measured. The interpretation of the remaining functions should be unambiguous.

4.2.5 Relational expressions

Relational expressions (⟨*rel_expr*⟩s) compare two arithmetic expressions (see Section 4.2.1 [Arithmetic expressions], page 48) or test an arithmetic expression for a property. A relational expression can be either TRUE or FALSE.

coNCePTuaL supports a variety of relational expressions. The following is the language's order of operations from highest to lowest precedence:

unary/	`IS EVEN`, `IS ODD`
binary/	'`=`', '`<`', '`>`', '`<=`', '`>=`', '`<>`', `DIVIDES`
ternary	`IS IN`
conjunctive	'`/\`'
disjunctive	'`\/`'

In addition, as in most programming languages, parentheses can be used to group subexpressions.

The unary relation `IS EVEN` is TRUE if a given arithmetic expression represents an even number and the unary relation `IS ODD` is TRUE if a given arithmetic expression represents an odd number. For example, '`456 IS EVEN`' is TRUE and '`64 MOD 6 IS ODD`' is FALSE.

The coNCePTuaL operators '`=`', '`<`', '`>`', '`<=`', '`>=`', and '`<>`' represent, respectively, the mathematical relations $=$, $<$, $>$, $\leq$, $\geq$, and $\neq$. These are all binary relations that operate on arithmetic expressions (see Section 4.2.1 [Arithmetic expressions], page 48). For example, '`2+2 = 4`' is TRUE and '`2**3 > 2**4`' is FALSE. The `DIVIDES` relation is TRUE if the first expression evenly divides the second, i.e., that $e_2 \equiv 0 \pmod{e_1}$. Hence, '`2`

`DIVIDES 1234`' (equivalent to '`1234 MOD 2 = 0`') is TRUE while '`2 DIVIDES 4321`' (equivalent to '`4321 MOD 2 = 0`') is FALSE.

The ternary relation `IS IN` has the form '⟨*expr*⟩ `IS IN [`⟨*expr*⟩`,` ⟨*expr*⟩`]`'. The relational expression '`x IS IN [a, b]`' is TRUE if x lies within the closed interval [a, b] and FALSE otherwise. The interval bounds a and b can be specified in any order. Hence, '`x IS IN [a, b]`' can be more precisely described as "$(a \leq x \leq b) \vee (b \leq x \leq a)$". To provide a few examples, '`4 IS IN [3,5]`' and '`4 IS IN [5,3]`' are both TRUE; '`5 IS IN [3,5]`' is TRUE; however, '`6 IS IN [3,5]`' is FALSE.

Conjunction ($\wedge$) and disjunction ($\vee$) combine multiple relational expressions. ⟨*rel_expr*⟩ '`/\`' ⟨*rel_expr*⟩ is TRUE if and only if both ⟨*rel_expr*⟩s are TRUE, and ⟨*rel_expr*⟩ '`\/`' ⟨*rel_expr*⟩ is TRUE if and only if either ⟨*rel_expr*⟩ is TRUE. For example, '`456 IS EVEN \/ 2**3 > 2**4`' is TRUE and '`456 IS EVEN /\ 2**3 > 2**4`' is FALSE. Conjunction and disjunction are both short-circuiting operations. Evaluation proceeds left-to-right. Expressions such as '`x<>0 /\ 1/x=1`' will therefore not result in a divide-by-zero error.

coNCePTuaL does not currently have a logical negation operator.

4.3 Task descriptions

Task descriptions are a powerful way of tersely describing the sources and targets of coNCePTuaL operations. Task IDs range from 0 to '`num_tasks-1`' (see Section A.2 [Predeclared variables], page 127). Operations involving out-of-bound task IDs are silently ignored.

As a side effect, a task description can declare a variable that can be used in subsequent expressions. (See Section 4.2 [Expressions], page 47.) There are two types of task descriptions: one for "source" tasks and one for "target" tasks. The two are syntactically similar but semantically different. Specifically, the scope of a variable declared in a ⟨*target_tasks*⟩ specification is more limited than one declared in a ⟨*source_task*⟩ specification.

Before introducing ⟨*source_task*⟩ and ⟨*target_tasks*⟩ specifications we first introduce the notion of a ⟨*restricted_ident*⟩, which is a variable declaration that can be used to define a set of tasks. We then present coNCePTuaL's complete set of mechanisms for describing sets of source and target tasks.

4.3.1 Restricted identifiers

A *restricted identifier* declares a variable, restricting it to the set of tasks that satisfy a given relational expression (see Section 4.2.5 [Relational expressions], page 60). The syntax is '⟨*ident*⟩ `SUCH THAT` ⟨*rel_expr*⟩' and represents the mathematical notion of "$\forall$⟨*ident*⟩$|$(⟨*rel_expr*⟩ $\wedge$ (0 $\leq$ ⟨*ident*⟩ $<$ ⟨#*tasks*⟩))". That is, "for all ⟨*ident*⟩ such that ⟨*rel_expr*⟩ is TRUE and ⟨*ident*⟩ is between zero and the number of tasks. . .".

As an example, '`evno SUCH THAT evno IS EVEN`' describes all even-numbered tasks. On each such task, the variable '`evno`' takes on that task's ID. Similarly, '`thr SUCH THAT 3 DIVIDES thr-1`' describes tasks 1, 4, 7, 10, 13. . . . On each of those tasks, '`thr`' will be bound to the task ID. On all other tasks, '`thr`' will be undefined. When order matters (as in the cases described in Section 4.4.2 [Sending], page 67 and Section 4.6.5 [Reordering task IDs], page 76), ⟨*ident*⟩ takes on task IDs in increasing order.

Note that an ⟨*ident*⟩ can be used anywhere that a ⟨*restricted_ident*⟩ is expected. In other words, the restricting ⟨*rel_expr*⟩ is optional and defaults to a TRUE expression.

4.3.2 Source tasks

A ⟨*source_task*⟩ specification takes one of four forms:

1. `ALL TASKS`
2. `ALL TASKS` ⟨*ident*⟩
3. `TASK` ⟨*expr*⟩
4. `TASKS` ⟨*restricted_ident*⟩

`ALL TASKS` specifies that each task will perform a given operation. If followed by a variable name (⟨*ident*⟩), each task will individually bind ⟨*ident*⟩ to its task ID—a number from zero to one less than the total number of tasks. That is, '`ALL TASKS me`' will bind '`me`' to '`0`' on task 0, '`1`' on task 1, and so forth.

'`TASK` ⟨*expr*⟩' specifies that only the task described by arithmetic expression ⟨*expr*⟩ will perform the given operation. For example, '`TASK 2*3+1`' says that only task 7 will act; the other tasks will do nothing.

'`TASKS` ⟨*restricted_ident*⟩' describes a set of tasks that will perform a given operation. For instance, '`TASKS x SUCH THAT x>0 /\ x<num_tasks-1`'—read as "tasks x such that x is greater than zero and x is less than *num_tasks* minus one"—expresses that a given operation should be performed on all tasks except the first and last in the computation. On each task that satisfies the relational expression, '`x`' will be bound to the task ID as in `ALL TASKS` above. Hence, '`x`' will be undefined on task 0, '`1`' on task 1, '`2`' on task 2, and so forth up to task '`num_tasks-1`', on which '`x`' will again be undefined.

As per the definitions in Section 4.1 [Primitives], page 46 and Section 4.3.1 [Restricted identifiers], page 61, respectively, ⟨*ident*⟩s and ⟨*restricted_ident*⟩s do not accept parentheses. Hence, '`TASKS (bad SUCH THAT bad IS EVEN)`' and '`ALL TASKS (no_good)`' result in parse errors while '`TASKS fine SUCH THAT fine IS EVEN`' and '`ALL TASKS dandy`' are acceptable constructs. As an analogy, '`x = 3`' is valid in many general-purpose programming languages while '`(x) = 3`' is not.

Variables declared in a '`source_task`' specification are limited in scope to the surrounding statement.

4.3.3 Target tasks

A ⟨*target_tasks*⟩ specification takes one of three forms:

1. `ALL OTHER TASKS`
2. `TASK` ⟨*expr*⟩
3. `TASKS` ⟨*restricted_ident*⟩

`ALL OTHER TASKS` is just like `ALL TASKS` in a ⟨*source_task*⟩ specification (see Section 4.3.2 [Source tasks], page 62) but applies to all tasks *except* the source task. Also, unlike `ALL TASKS`, `ALL OTHER TASKS` does not accept an ⟨*ident*⟩ term.

'`TASK` ⟨*expr*⟩' specifies that only the task described by arithmetic expression ⟨*expr*⟩ is the target of the given operation. ⟨*expr*⟩ can use a variable declared as a ⟨*source_task*⟩.

For example, if ⟨*source_task*⟩ is '`ALL TASKS x`', then a ⟨*target_tasks*⟩ of '`TASK (x+1) MOD num_tasks`' refers to each task's right neighbor (with wraparound from `num_tasks` to '`0`').

'`TASKS` ⟨*restricted_ident*⟩' describes a set of tasks that will perform a given operation. As with its '`source_task`' counterpart, a ⟨*restricted_ident*⟩ declares a variable. However, in a ⟨*target_tasks*⟩ specification the variable's scope is limited to the relational expression within the ⟨*restricted_ident*⟩. As an example, '`TASKS dst SUCH THAT dst>src`' refers to all tasks '`dst`' with a greater ID than a (previously declared) task '`src`'.

4.4 Communication statements

Communication statements are the core of any coNCePTuaL program. The coNCePTuaL language makes it easy to express a variety of communication features:

- synchronous or asynchronous communication
- unaligned, aligned (to arbitrary byte boundaries), or misaligned (from a page boundary) message buffers
- ignored, touched, or verified message contents
- unique or recycled message buffers
- point-to-point or collective operations

Communication statements are performed by an arbitrary ⟨*source_task*⟩ (see Section 4.3.2 [Source tasks], page 62) and may involve arbitrary ⟨*target_tasks*⟩ (see Section 4.3.3 [Target tasks], page 62). After explaining how to describe a message to coNCePTuaL (see Section 4.4.1 [Message specifications], page 63) this section presents each communication statement in turn and explains its purpose, syntax, and semantics.

4.4.1 Message specifications

A *message specification* describes a set of messages. The following is a formal definition:

⟨*message_spec*⟩ ::= ⟨*item_count*⟩
[`NONUNIQUE | UNIQUE`]
⟨*item_size*⟩
[`UNALIGNED |`
⟨*message_alignment*⟩ `ALIGNED |`
⟨*message_alignment*⟩ `MISALIGNED`]
`MESSAGES`
[`WITH VERIFICATION | WITH DATA TOUCHING |`
`WITHOUT VERIFICATION | WITHOUT DATA TOUCHING`]
[`FROM BUFFER` ⟨*expr*⟩ `|`
`FROM THE DEFAULT BUFFER`]

Within a `RECEIVE` statement (see Section 4.4.3 [Receiving], page 68), a ⟨*message_spec*⟩'s `FROM` keyword must be replaced with `INTO`.

A `SEND` statement's `WHO RECEIVES IT` clause (see Section 4.4.2 [Sending], page 67) utilizes a slightly different message specification, which is referred to here as a ⟨*recv_message_spec*⟩:

⟨*recv_message_spec*⟩ ::= [`SYNCHRONOUSLY | ASYNCHRONOUSLY`]
[`AS [A|AN]`]

```
[NONUNIQUE | UNIQUE]
[UNALIGNED |
 ⟨message_alignment⟩ ALIGNED |
 ⟨message_alignment⟩ MISALIGNED]
MESSAGES]
[WITH VERIFICATION | WITH DATA TOUCHING |
 WITHOUT VERIFICATION | WITHOUT DATA TOUCHING]
[FROM BUFFER ⟨expr⟩ |
 FROM THE DEFAULT BUFFER]
```

We now describe in turn each component of a ⟨*message_spec*⟩ and ⟨*recv_message_spec*⟩.

Item count

The ⟨*item_count*⟩ says how many messages the ⟨*message_spec*⟩ represents. It can be an ⟨*expr*⟩ or one or `A` or `AN`, indicating '`1`'.

Unique messages

Normally, the coNCePTuaL backends recycle message memory to reduce the program's memory requirements and improve performance. By adding the keyword `UNIQUE`, every message buffer will reside in a unique memory region. `NONUNIQUE` explicitly specifies the default, buffer-recycling behavior.

Item size

The message size is represented by the ⟨*item_size*⟩ nonterminal. It can be expressed in one of two ways:

⟨*item_size*⟩ ::= ⟨*expr*⟩ ⟨*data_multiplier*⟩
| ⟨*data_type*⟩ `SIZED`

A ⟨*data_multiplier*⟩ is a scaling factor that converts a unitless number into a number of bytes. The following are the valid possibilities for ⟨*data_multiplier*⟩ and the number of bytes which which they multiply ⟨*expr*⟩:

`BIT` 1/8 bytes, rounded up to the nearest integral number of bytes

`BYTE` 1 byte

`HALFWORD` 2 bytes

`WORD` 4 bytes

`INTEGER` the number of bytes in the backend's fundamental integer type

`DOUBLEWORD` 8 bytes

`QUADWORD` 16 bytes

`PAGE` the number of bytes in an operating-system page

`KILOBYTE` 1,024 bytes

`MEGABYTE` 1,048,576 bytes

`GIGABYTE` 1,073,741,824 bytes

A ⟨*data_type*⟩ is an "atomic" unit of data. It can be any of the following:

`BYTE` 1 byte

`HALFWORD` 2 bytes

`WORD` 4 bytes

`INTEGER` the number of bytes in the backend's fundamental integer type

`DOUBLEWORD`
8 bytes

`QUADWORD` 16 bytes

`PAGE` the number of bytes in an operating-system page

Hence, valid ⟨*item_size*⟩s include, for example, '`16 MEGABYTE`' or '`PAGE SIZED`'. Note that `INTEGER` varies in size based on the backend, backend compiler, and CPU architecture (but is commonly either 4 or 8 bytes); `PAGE` varies in size from operating system to operating system; each of the other ⟨*data_type*⟩s has a fixed size, as indicated above.

Message alignment

Messages are normally allocated with arbitrary alignment in memory. However, coNCePTuaL can force a specific alignment relative to the operating-system page size (commonly 4 KB or 8 KB, but significantly larger sizes are gaining popularity). ⟨*message_alignment*⟩ is expressed as either a ⟨*data_type*⟩ (described above under "Item size") or as an ⟨*expr*⟩ (see Section 4.2.1 [Arithmetic expressions], page 48) followed by a ⟨*data_multiplier*⟩ (also described above under "Item size"). '`64 BYTE`', '`3 MEGABYTE`', and '`QUADWORD`' are therefore all valid examples of ⟨*message_alignment*⟩s. Bit counts are rounded up to the nearest byte count, so '`27 BITS`' is actually equivalent to '`4 BYTES`'.

The `ALIGNED` keyword forces coNCePTuaL to align messages on *exactly* the specified alignment. Hence, a '`HALFWORD ALIGNED`' message can begin at memory locations 0, 2, 4, 6, 8, ..., $2k$ $(k \in \mathbb{Z}^+)$. In contrast, the `MISALIGNED` keyword forces coNCePTuaL to align messages the given number of bytes (positive or negative) past a page boundary. For example, if pages are 8192 bytes in size then a message described as '`HALFWORD MISALIGNED`' can begin at memory locations 2, 8194, 16386, 24578, ..., $8192k + 2$ $(k \in \mathbb{Z}^+)$. Unlike `ALIGNED`, `MISALIGNED` supports negative alignments. If the page size is 4096 bytes, then '`-10 BYTE MISALIGNED`' enables a message to begin at memory locations 4086, 8182, 12278, etc. The `MISALIGNED` alignment is taken modulo the page size. Therefore, with a 4096-byte page size, '`10000 BYTE MISALIGNED`' is the same as '`1808 BYTE MISALIGNED`'.

The `UNALIGNED` keyword explicitly specifies the default behavior, with messages aligned on arbitrary boundaries.

Data touching

A ⟨*message_spec*⟩ described as being `WITH DATA TOUCHING` will force every word in a message to be both read and written ("touched"). When ⟨*message_spec*⟩ describes an outgoing message, the data will be touched before transmission. When ⟨*message_spec*⟩

describes an incoming message, the data will be touched after reception. In a sense, `WITH DATA TOUCHING` presents a more realistic assessment of network performance, as real applications almost always access the data they send or receive. It also distinguishes between messaging layers that implicitly touch data and those that can transmit data without having to touch it. One would expect the latter to perform better when the data is not touched, as the former may be paying a penalty for touching the data. However, either could perform better when messages are sent `WITH DATA TOUCHING`, because the latter now has to pay the penalty that the former has already paid.

Another form of data-touching supported by coNCePTuaL is `WITH VERIFICATION`. This causes the source task to write known, but randomly generated, data into the message before transmission and the target task to verify that every bit was correctly received. When a message is received `WITH VERIFICATION`, the `bit_errors` variable (see Section A.2 [Predeclared variables], page 127) is updated appropriately.

`WITHOUT DATA TOUCHING` and `WITHOUT VERIFICATION` are synonymous. Both explicitly specify the default behavior of neither touching nor verifying message contents.

Buffer control

The coNCePTuaL run-time library allocates a unique message buffer for each message sent/received with the `UNIQUE` keyword. The message buffers for `NONUNIQUE` messages are recycled subject to the constraint that no two concurrent transmissions will reference the same buffer. For example, if a task performs a synchronous send followed by a synchronous receive, those operations must be executed serially and will therefore share a message buffer. If, instead, a task performs an asynchronous send followed by an asynchronous receive, those operations may overlap, so coNCePTuaL will use different message buffers for the two operations.

Message specifications enable the programmer to override the default buffer-allocation behavior. If a message is sent `FROM BUFFER` ⟨*expr*⟩ or received `INTO BUFFER` ⟨*expr*⟩, the message is guaranteed to be sent/received using the specified buffer number. For example, `FROM BUFFER` and `INTO BUFFER` can be used to force a synchronous send and synchronous receive to use different buffers or an asynchronous send and asynchronous receive to use the same buffer. If ⟨*expr*⟩ is negative, the behavior is the same as if `FROM BUFFER`/`INTO BUFFER` was not specified. `FROM THE DEFAULT BUFFER` and `INTO THE DEFAULT BUFFER` also explicitly specify the default buffer-allocation behavior.

Blocking semantics

By default—or if `SYNCHRONOUSLY` is specified—messages are sent synchronously. That is, a sender blocks (i.e., waits) until the message buffer is safe to reuse before it continues and a receiver blocks until it actually receives the message. The `ASYNCHRONOUSLY` keyword specifies that messages should be sent and received asynchronously. That is, the program merely posts the message (i.e., declares that it should eventually be sent and/or received) and immediately continues executing. Asynchronous messages must be *completed* as described in Section 4.4.4 [Awaiting completion], page 69.

4.4.2 Sending

The SEND statement is fundamental to coNCePTuaL. It is used to send a multiple messages from multiple source tasks to multiple target tasks. The syntax is formally specified as follows:

⟨*send_stmt*⟩ ::= ⟨*source_task*⟩
[ASYNCHRONOUSLY] SENDS
⟨*message_spec*⟩
TO [UNSUSPECTING] ⟨*target_tasks*⟩
| ⟨*source_task*⟩
[ASYNCHRONOUSLY] SENDS
⟨*message_spec*⟩
TO ⟨*target_tasks*⟩
WHO RECEIVE IT
⟨*recv_message_spec*⟩

⟨*source_task*⟩ is described in Section 4.3.2 [Source tasks], page 62; ⟨*message_spec*⟩ and ⟨*recv_message_spec*⟩ are described in Section 4.4.1 [Message specifications], page 63; and, ⟨*target_tasks*⟩ is described in Section 4.3.3 [Target tasks], page 62.

The SEND statement's simplest form, "⟨*source_task*⟩ SENDS ⟨*message_spec*⟩ TO ⟨*target_tasks*⟩", is fairly straightforward. The following is a example:

```
TASK 0 SENDS A 0 BYTE MESSAGE TO TASK 1
```

The only subtlety in the preceding statement is that it implicitly causes task 1 to perform a corresponding receive. This receive can be suppressed by adding the keyword UNSUSPECTING before the ⟨*target_tasks*⟩ description:

```
TASK 0 SENDS A 0 BYTE MESSAGE TO UNSUSPECTING TASK 1
```

Here are some further examples of valid ⟨*send_stmt*⟩s:

- `ALL TASKS SEND A 64 KILOBYTE MESSAGE TO TASK 0`
- `TASK num_tasks-1 SENDS 5 53 BYTE PAGE ALIGNED MESSAGES TO ALL OTHER TASKS`
- `TASKS upper SUCH THAT upper>=num_tasks/2 ASYNCHRONOUSLY SEND A 0 BYTE MESSAGE TO TASK upper/2`
- `TASKS nonzero SUCH THAT nonzero>0 SEND nonzero 1E3 BYTE MESSAGES TO UNSUSPECTING TASK 0`

One subtlety of the SEND statement when used without UNSUSPECTING involves the orderings of the sends and receives. The rule is that receives are posted before sends. Furthermore, ⟨*restricted_ident*⟩s (see Section 4.3.1 [Restricted identifiers], page 61) are evaluated in order from 0 to *num_tasks* − 1. The implication is that a statement such as '`TASKS ev SUCH THAT ev IS EVEN /\ ev<6 SEND A 4 WORD MESSAGE TO TASK ev+2`' is exactly equivalent to the following ordered sequence of statements (assuming *num_tasks* ≥ 5):

1. `TASK 2 RECEIVES A 4 WORD MESSAGE FROM TASK 0`
2. `TASK 4 RECEIVES A 4 WORD MESSAGE FROM TASK 2`
3. `TASK 6 RECEIVES A 4 WORD MESSAGE FROM TASK 4`
4. `TASK 0 SENDS A 4 WORD MESSAGE TO UNSUSPECTING TASK 2`
5. `TASK 2 SENDS A 4 WORD MESSAGE TO UNSUSPECTING TASK 4`

6. `TASK 4 SENDS A 4 WORD MESSAGE TO UNSUSPECTING TASK 6`

(The `RECEIVE` statement is described in Section 4.4.3 [Receiving], page 68.)

If the above sequence were executed, tasks 2, 4, and 6 would immediately block on their receives (steps 1–3). Task 0 would awaken task 2 by sending it a message (step 4). Then, task 2 would be able to continue to step 5 at which point it would send a message to task 4. Task 4 would then finally be able to send a message to task 6 (step 6). Hence, even though the original coNCePTuaL statement encapsulates multiple communication operations, the component communications proceed sequentially because of data dependences and because the operations are blocking.

As should now be apparent, there are a number of attributes associated with every message transmission:

- synchronous vs. asynchronous operation
- unique vs. recycled message buffers
- unaligned vs. aligned vs. misaligned message buffers
- no data touching vs. data touching vs. data verification
- implicit vs. explicit message-buffer selection

When `UNSUSPECTING` is omitted, the implicit `RECEIVE` statement normally inherits all of the attributes of the corresponding `SEND`. However, the second form of a ⟨*send_stmt*⟩, which contains a `WHO RECEIVES IT` (or `WHO RECEIVES THEM`) clause, enables the receiver's attributes to be overridden on a per-attribute basis. For instance, consider the following `SEND` statement:

```
TASK 0 SENDS A 1 MEGABYTE MESSAGE TO TASK 1 WHO RECEIVES IT
ASYNCHRONOUSLY
```

The alternative sequence of statements that does not use `WHO RECEIVES IT` is less straightforward:

1. `TASK 1 ASYNCHRONOUSLY RECEIVES A 1 MEGABYTE MESSAGE FROM TASK 0`
2. `TASK 0 SENDS A 1 MEGABYTE MESSAGE TO UNSUSPECTING TASK 1`

Some further examples of `WHO RECEIVES IT` follow:

```
TASKS left SUCH THAT left IS EVEN SEND 5 2 KILOBYTE 64 BYTE ALIGNED
MESSAGES TO TASKS left+1 WHO RECEIVE THEM AS UNALIGNED MESSAGES WITH
DATA TOUCHING

TASK num_tasks-1 ASYNCHRONOUSLY SENDS A 1E5 BYTE MESSAGE WITH
VERIFICATION TO TASK 0 WHO RECEIVES IT SYNCHRONOUSLY

TASK leaf SUCH THAT KNOMIAL_CHILDREN(leaf,2)=0 SENDS A UNIQUE 1536
BYTE MESSAGE WITH DATA TOUCHING TO TASK KNOMIAL_PARENT(leaf,2) WHO
RECEIVES IT ASYNCHRONOUSLY AS A NONUNIQUE QUADWORD ALIGNED MESSAGE
WITHOUT DATA TOUCHING INTO BUFFER KNOMIAL_PARENT(leaf,2)
```

4.4.3 Receiving

Section 4.4.2 [Sending], page 67, mentioned the ⟨*send_stmt*⟩'s `UNSUSPECTING` keyword, which specifies that the targets should not implicitly perform a receive operation. Because

every send must have a matching receive, coNCePTuaL offers a RECEIVE statement which explicitly receives a set of messages. A ⟨*receive_stmt*⟩ is much like a ⟨*send_stmt*⟩ (see Section 4.4.2 [Sending], page 67) with the ⟨*source_task*⟩ and ⟨*target_tasks*⟩ in the reverse order:

⟨*receive_stmt*⟩ ::= ⟨*target_tasks*⟩
[ASYNCHRONOUSLY] RECEIVE
⟨*message_spec*⟩
FROM ⟨*source_task*⟩

⟨*target_tasks*⟩ is described in Section 4.3.3 [Target tasks], page 62; ⟨*message_spec*⟩ is described in Section 4.4.1 [Message specifications], page 63; and, ⟨*source_task*⟩ is described in Section 4.3.2 [Source tasks], page 62.

For each message sent via a SEND...TO UNSUSPECTING statement there must be a RECEIVE statement that receives a message of the same size. The ⟨*target_tasks*⟩'s ⟨*message_spec*⟩ can, however, specify different values for message uniqueness, message alignment, and data touching. In addition, the source and target do not need to agree on the use of the ASYNCHRONOUSLY keyword. The only restriction is that WITH VERIFICATION will return spurious results if used by the target but not by the source. Hence, the following ⟨*send_stmt*⟩ and ⟨*receive_stmt*⟩ correctly match each other:

```
TASK 0 SENDS 3 4 KILOBYTE MESSAGES TO UNSUSPECTING TASK 1

TASK 1 ASYNCHRONOUSLY RECEIVES 3 UNIQUE 4 KILOBYTE 48 BYTE ALIGNED
MESSAGES WITH DATA TOUCHING FROM TASK 0.
```

In general, it is better to use a single SEND statement with a WHO RECEIVES IT clause (see Section 4.4.2 [Sending], page 67) than a RECEIVE plus a matching SEND...TO UNSUSPECTING; the former is less error-prone than the latter. However, the latter is useful for programs in which a set of receives is posted, then the tasks perform various communication, computation, and synchronization operations, and—towards the end of the program—the matching sends are posted. That sort of split-phase structure requires separate SEND and RECEIVE statements.

4.4.4 Awaiting completion

When a message is sent or received asynchronously it must eventually be *completed*. In some messaging layers, asynchronous messages are not even sent or received until completion time. coNCePTuaL provides the following statement for completing messages that were send/received asynchronously:

⟨*wait_stmt*⟩ ::= ⟨*source_task*⟩
AWAITS COMPLETION

That is, a ⟨*wait_stmt*⟩ simply specifies the set of tasks that should block until all of their pending communications complete. ⟨*source_task*⟩ is as defined in Section 4.3.2 [Source tasks], page 62. Note that a ⟨*wait_stmt*⟩ blocks until *all* pending communications complete. coNCePTuaL does not provide finer-grained control over completions. It is safe, however, for a task to AWAIT COMPLETION even if it has no asynchronous messages pending.

4.4.5 Multicasting

Although a single ⟨*send_stmt*⟩ (see Section 4.4.2 [Sending], page 67) can specify multiple messages at once, these messages are sent one at a time. *Multicasting* is a form of collective communication in which a set of tasks collaborates to deliver a message from a source to multiple targets. On many systems, multicasting a message to N tasks is more efficient than sending a sequence of N individual messages. coNCePTuaL supports multicasting as follows:

⟨*mcast_stmt*⟩ ::= ⟨*source_task*⟩
[`ASYNCHRONOUSLY`] `MULTICASTS`
⟨*message_spec*⟩
`TO` ⟨*target_tasks*⟩

Unlike ⟨*send_stmt*⟩s, ⟨*mcast_stmt*⟩s do not support the `UNSUSPECTING` keyword. This is because `MULTICASTS` is a collective operation: all parties are active participants in delivering messages to the ⟨*target_tasks*⟩.

⟨*source_task*⟩ (see Section 4.3.2 [Source tasks], page 62) and ⟨*target_tasks*⟩ (see Section 4.3.3 [Target tasks], page 62) can be either disjoint or overlapping sets. That is, either of the following is legal:

```
TASK 0 MULTICASTS A 16 BYTE MESSAGE TO TASKS recip SUCH THAT recip<4
TASK 0 MULTICASTS A 16 BYTE MESSAGE TO TASKS recip SUCH THAT recip>=4
```

Note that in the first ⟨*mcast_stmt*⟩, task 0 both sends and receives a message, while in the second ⟨*mcast_stmt*⟩, task 0 sends but does not receive.

4.4.6 Synchronizing

coNCePTuaL enables sets of tasks to perform *barrier synchronization*. The semantics are that no task can finish synchronizing until all tasks have started synchronizing. The syntax is as follows:

⟨*sync_stmt*⟩ ::= ⟨*source_task*⟩
`SYNCHRONIZES`

A ⟨*sync_stmt*⟩ can be used to ensure that one set of statements has completed before beginning another set. For example, a coNCePTuaL program might have a set of tasks post a series of asynchronous receives (see Section 4.4.3 [Receiving], page 68), then make '`ALL TASKS SYNCHRONIZE`' before having another set of tasks perform the corresponding `UNSUSPECTING` sends (see Section 4.4.2 [Sending], page 67). This procedure ensures that all of the target tasks are ready to receive before the source tasks start sending to them.

4.5 I/O statements

coNCePTuaL provides two statements for presenting information. One statement writes simple messages to the standard output device and is intended to be used for providing status information during the run of a program. The other statement provides a powerful mechanism for storing performance and correctness data to a log file.

4.5.1 Utilizing log-file comments

⟨*output_stmt*⟩s and ⟨*log_stmt*⟩s have limited access to the ⟨*key*:*value*⟩ pairs that are written as comments at the top of every log file as shown in Section 3.4.1 [Log-file format], page 23. Given a key, *key*, the string expression '`THE VALUE OF` *key*' represents the value associated with that key or the empty string if *key* does not appear in the log-file comments:

⟨*string_or_log_comment*⟩ ::= ⟨*string*⟩
| `THE VALUE OF` ⟨*string*⟩

That is, '`"CPU frequency"`' means the literal string "CPU frequency" while '`THE VALUE OF "CPU frequency"`' translates to string like "1300000000 Hz (1.3 GHz)". Environment variables are also considered keys and are therefore acceptable input to a `THE VALUE OF` construct.

4.5.2 Writing to standard output

coNCePTuaL's `OUTPUT` keyword is used to write a message from one or more source tasks (see Section 4.3.2 [Source tasks], page 62) to the standard output device. This is useful for providing progress reports during the execution of long-running coNCePTuaL programs. An ⟨*output_stmt*⟩ looks like this:

⟨*output_stmt*⟩ ::= ⟨*source_task*⟩
`OUTPUTS`
⟨*expr*⟩ | ⟨*string_or_log_comment*⟩
[`AND` ⟨*expr*⟩ | ⟨*string_or_log_comment*⟩]*

The following are some sample ⟨*output_stmt*⟩s:

```
TASK 0 OUTPUTS "Hello, world!"

TASKS nr SUCH THAT nr>0 OUTPUT nr AND "'s parent is " AND nr>>1 AND
" and its children are " AND nr<<1 AND " and " AND nr<<1+1

ALL TASKS me OUTPUT "Task " AND me AND " is running on host " AND THE
VALUE OF "Host name" AND " and plans to send to task " AND (me+1) MOD
num_tasks
```

`OUTPUT` does not implicitly output spaces between terms. Hence, '`OUTPUT "Yes" AND "No"`' will output "YesNo", not "Yes No". Also, `AND` binds tighter as a binary operator (see Section 4.2.1 [Arithmetic expressions], page 48) than within an ⟨*output_stmt*⟩. Consequently, '`OUTPUT 6 AND 3`' will produce "2", not "63". Although it is unlikely that a program would ever need to output two arithmetic expressions with no intervening text, an empty string can be used for this purpose: '`OUTPUT 6 AND "" AND 3`'.

An ⟨*output_stmt*⟩ implicitly outputs a newline character at the end. Additional newline characters can be output by embedding '`\n`' in a string. (see Section 4.1 [Primitives], page 46.) coNCePTuaL does not provide a means for suppressing the newline, however.

4.5.3 Writing to a log file

After performing a network correctness or performance test it is almost always desirable to store the results in a file. coNCePTuaL has language support for writing tabular data to a log file. The ⟨*log_stmt*⟩ command does the bulk of the work:

⟨*log_stmt*⟩ ::= ⟨*source_task*⟩
LOGS
⟨*aggr_expr*⟩ AS ⟨*string_or_log_comment*⟩
[AND ⟨*aggr_expr*⟩ AS ⟨*string_or_log_comment*⟩]*

The idea behind a ⟨*log_stmt*⟩ is that a set of source tasks (see Section 4.3.2 [Source tasks], page 62) log an aggregate expression (see Section 4.2.3 [Aggregate expressions], page 59) to a log file under the column heading ⟨*string_or_log_comment*⟩. Each task individually maintains a separately named log file so there is no ambiguity over which task wrote which entries.

Each (static) LOGS statement in a coNCePTuaL program specifies one or more columns of the log file. Every dynamic execution of a LOGS statement writes a single row to the log file. A single LOGS statement should suffice for most coNCePTuaL programs.

The following are some examples of ⟨*log_stmt*⟩s:

```
ALL TASKS LOG bit_errors AS "Bit errors"

TASK 0 LOGS THE msgsize AS "Bytes" AND
            THE MEDIAN OF (1E6*bytes_sent)/(1M*elapsed_usecs) AS "MB/s"
```

The first example produces a log file like the following:

```
"Bit errors"
"(all data)"
3
```

The second example produces a log file like this:

```
"Bytes","MB/s"
"(only value)","(median)"
65536,179.9416266
```

Note that in each log file, the coNCePTuaL run-time system writes two rows of column headers for each column. The first row contains ⟨*string_or_log_comment*⟩ as is. The second row describes the ⟨*aggr_func*⟩ (see Section 4.2.4 [Aggregate functions], page 60) used to aggregate the data. One or more rows of data follow.

Assume that the second ⟨*log_stmt*⟩ presented above appears within a loop (see Section 4.7.2 [Iterating], page 79). It is therefore important to include the THE keyword before 'msgsize' to assert that the expression 'msgsize' is constant across invocations of the ⟨*log_stmt*⟩ and that, consequently, only a single row of data should be written to the log file. Using 'msgsize' without the THE would produce a column of data with one row per ⟨*log_stmt*⟩ invocation:

```
"Bytes","MB/s"
"(all data)","(median)"
65536,179.9416266
65536,
65536,
65536,
65536,
     .
     .
     .
```

The rules that determine how `LOGS` statements produce rows and columns of a log file are presented below:

1. Each *static* `LOGS` statement (and `AND` clause within a `LOGS` statement) in a program produces a unique column.
2. Each *dynamic* execution of a `LOGS` statement appends a row to the column(s) it describes.
3. Each top-level complex statement (see Section 4.9 [Complete programs], page 87) produces a new table in the log file.

Note that the choice of column name is inconsequential for determining what columns are written to the log file:

```
TASK 0 LOGS 314/100 AS "Pi" AND 22/7 AS "Pi"
```

```
"Pi","Pi"
"(all data)","(all data)"
3.14,3.142857143
```

Computing aggregates

What if '`msgsize`' takes on a number of values throughout the execution of the program and for each value a number of runs is performed? How would one log the median of each set of data? Using '`THE msgsize`' won't work because the message size is not constant. Using '`msgsize`' alone won't work either because coNCePTuaL would then take the median of the times gathered across *all* message sizes, which is undesirable. The solution is for the program to specify explicitly when aggregate functions (`MEDIAN` and all of the other functions listed in Section 4.2.4 [Aggregate functions], page 60) compute a value:

⟨*flush_stmt*⟩ ::= ⟨*source_task*⟩
`COMPUTES AGGREGATES`

The intention is that an inner loop might `LOG` data after every iteration and an outer loop would '`COMPUTE AGGREGATES`' after each iteration.

4.6 Other statements

coNCePTuaL contains a few more statements than those described in Section 4.4 [Communication statements], page 63 and Section 4.5 [I/O statements], page 70. As there is no category that clearly describes the remaining statements, they are listed here in this "catch-all" section.

4.6.1 Resetting counters

It is common for benchmarks to perform a few warmup iterations before beginning the actual test. This hides the time needed initially to establish connections, populate the various processor and network caches, etc., making subsequent measurements more uniform. However, these warmup iterations, like everything else in coNCePTuaL, exhibit the side effect of updating coNCePTuaL's dynamic variables (see Section A.2 [Predeclared variables], page 127) such as `elapsed_usecs` and `msgs_sent`. The solution is to reset all of those dynamic variables to zero before beginning the core part of the program. The `RESET` statement does just that:

⟨*reset_stmt*⟩ ::= ⟨*source_task*⟩
`RESETS ITS COUNTERS`

Hence, writing '`ALL TASKS RESET THEIR COUNTERS`' causes each task to reset all of the variables listed in Section A.2 [Predeclared variables], page 127—with the exception of `num_tasks`—to zero. Note that `ITS` and `THEIR`, like `RESET` and `RESETS`, are considered synonyms (see Section 4.1 [Primitives], page 46).

4.6.2 Asserting conditions

coNCePTuaL programs can encode the run-time conditions that must hold in order for the test to run properly. This is achieved through *assertions*, which are expressed as follows:

⟨*assert_stmt*⟩ ::= `ASSERT THAT` ⟨*string*⟩
`WITH` ⟨*rel_expr*⟩

⟨*string*⟩ is a message to be reported to the user if the assertion fails. ⟨*rel_expr*⟩ is a relational expression (as described in Section 4.2.5 [Relational expressions], page 60) that must evaluate to TRUE for the program to continue running. Assertion failures are considered fatal errors. They cause the coNCePTuaL program to abort immediately.

Here are some sample ⟨*assert_stmt*⟩s:

```
ASSERT THAT "the bandwidth test requires at least two tasks" WITH
num_tasks >= 2

ASSERT THAT "pairwise ping-pongs require an even number of task"
WITH num_tasks IS EVEN

ASSERT THAT "this program requires a square number of tasks" WITH
SQRT(num_tasks)**2 = num_tasks
```

(For the last example, recall that coNCePTuaL expressions are of integer type. Hence, the example's ⟨*rel_expr*⟩ is mathematically equivalent to $\lfloor\sqrt{N}\rfloor^2 = N$, which is TRUE if and only if N is a square.)

4.6.3 Delaying execution

It is sometimes interesting to measure the progress of a communication pattern when delays are inserted at various times on various tasks. coNCePTuaL provides two mechanisms for inserting delays: one that relinquishes the CPU while delaying (`SLEEP`) and one that hoards it (`COMPUTE`).

⟨*delay_stmt*⟩ ::= ⟨*source_task*⟩
SLEEPS | COMPUTES
FOR ⟨*expr*⟩ ⟨*time_unit*⟩

⟨*source_task*⟩ (see Section 4.3.2 [Source tasks], page 62) specifies the set of tasks that will stall. ⟨*time_unit*⟩ must be one of the following keywords:

- MICROSECONDS
- MILLISECONDS
- SECONDS
- MINUTES
- HOURS
- DAYS

⟨*expr*⟩ (see Section 4.2.1 [Arithmetic expressions], page 48) specifies the number of ⟨*time_unit*⟩s for which to delay.

Delay times are only approximate. SLEEP's accuracy depends upon the operating-system's clock resolution or length of time quantum (commonly measured in milliseconds or tens of milliseconds). COMPUTE, which is implemented by repeatedly reading a variable until the desired amount of time elapses, is calibrated during the coNCePTuaL run-time system's initialization phase and can be adversely affected by intermittant system load. Both forms of ⟨*delay_stmt*⟩ attempt to measure wall-clock time ("real time"), not just the time the program is running ("virtual time"). Because the delay times are approximate, it is strongly recommended that the elapsed_usecs variable (see Section A.2 [Predeclared variables], page 127) be employed to determine the actual elapsed time.

4.6.4 Touching memory

While the statements described in Section 4.6.3 [Delaying execution], page 74 delay for a specified length of time, it is also possible to delay for the duration of a specified amount of "work". "Work" is expressed in terms of memory accesses. That is, a coNCePTuaL program can touch (i.e., read plus write) data with a given stride from a memory region of a given size. By varying these parameters, a program can emulate an application's computation by hoarding the CPU or any level of the memory hierarchy.

⟨*touch_stmt*⟩ ::= ⟨*source_task*⟩
TOUCHES
[⟨*expr*⟩ ⟨*data_type*⟩ OF]
AN ⟨*item_size*⟩ MEMORY REGION
[⟨*expr*⟩ TIMES]
[WITH STRIDE ⟨*expr*⟩ ⟨*data_type*⟩ | WITH RANDOM STRIDE]

⟨*item_size*⟩ and ⟨*data_type*⟩ are described in Section 4.4.1 [Message specifications], page 63 and ⟨*expr*⟩ is described in Section 4.2.1 [Arithmetic expressions], page 48.

As shown by the formal definition of ⟨*touch_stmt*⟩ the required components are a ⟨*source_task*⟩ and the size of the memory region to touch. By default, every WORD (see Section 4.4.1 [Message specifications], page 63) of memory in the region is touched exactly once. The type of data that is touched can be varied with an '⟨*expr*⟩ ⟨*data_type*⟩ OF' clause. For instance, '100 BYTES OF' of a memory region will touch individual bytes. An optional

repeat count enables the memory region (or subset thereof) to be touched multiple times. Hence, if '`TASK 0 TOUCHES A 6 MEGABYTE MEMORY REGION 5 TIMES`', then the touch will be performed as if '`TASK 0`' were told to '`TOUCH 5*6M BYTES OF A 6 MEGABYTE MEMORY REGION 1 TIME`' or simply to '`TOUCH 5*6M BYTES OF A 6 MEGABYTE MEMORY REGION`'.

By default, every ⟨*data_type*⟩ of data is touched. However, a ⟨*touch_stmt*⟩ provides for touching only a subset of the ⟨*data_type*⟩s in the memory region. By writing '`WITH STRIDE` ⟨*expr*⟩ ⟨*data_type*⟩', only the first ⟨*data_type*⟩ out of every ⟨*expr*⟩ will be touched. Instead of specifying an exact stride, the memory region can be accessed in random order using the `WITH RANDOM STRIDE` clause.

Unless the number of touches and data type are specified explicitly, the number of `WORD`s that are touched is equal to the size of the memory region divided by the stride length then multiplied by the repeat count. Therefore, if '`TASK 0 TOUCHES AN 8 MEGABYTE MEMORY REGION 2 TIMES WITH STRIDE 8 WORDS`', then a total of $(2^{23}/(4 \times 8)) \times 2 = 524288$ touches will be performed. For the purpose of the preceding calculation, '`WITH RANDOM STRIDE`' should be treated as if it were '`WITH STRIDE 1 WORD`' (again, unless the number of touches and data type are specified explicitly).

To save memory, all `TOUCH` statements in a coNCePTuaL program access subsets of the same region of memory, whose size is determined by the maximum needed. However, each dynamic execution of a ⟨*touch_stmt*⟩ starts touching from where the previous execution left off. For example, consider the following statement:

```
TASK 0 TOUCHES 100 WORDS OF A 200 WORD MEMORY REGION
```

The first time that that statement is executed within a loop (see Section 4.7.2 [Iterating], page 79), the first 200 words are touched. The second time, the second 200 words are touched. The third time, the index into the region wraps around and the first 200 words are touched again.

Each static ⟨*touch_stmt*⟩ maintains its own index into the memory region. Therefore, the first of the following two statements will terminate successfully (assuming it's not executed in the body of a loop) while the second will result in a run-time error because the final byte of the final word does not fit within the given memory region.

```
TASK 0 TOUCHES 100 WORDS OF A 799 BYTE MEMORY REGION THEN
TASK 0 TOUCHES 100 WORDS OF A 799 BYTE MEMORY REGION

FOR 2 REPETITIONS TASK 0 TOUCHES 100 WORDS OF A 799 BYTE MEMORY REGION
```

(`THEN` is described in Section 4.7.1 [Combining statements], page 78, and `FOR... REPETITIONS` is described in Section 4.7.2 [Iterating], page 79.) The first statement shown above touches the same 100 words (400 bytes) in each of the two ⟨*touch_stmt*⟩s. The second statement touches the first 100 words the first time the ⟨*touch_stmt*⟩ is executed and fails when trying to touch the (only partially extant) second 100 words.

4.6.5 Reordering task IDs

coNCePTuaL distinguishes between "task IDs", which are used in task descriptions (see Section 4.3 [Task descriptions], page 61) and "processor IDs", which are assigned by the underlying communication layer. As stated in Section 3.3 [Running coNCePTuaL programs], page 21, a coNCePTuaL program has no control over how processor IDs map to physical processors. It therefore has no way to specify, for instance, that a set of tasks must

run on the same multiprocessor node (or on different nodes, for that matter). Initially, every task's task ID is set equal to its processor ID. However, while processor IDs are immutable, task IDs can be changed dynamically during the execution of a program. Altering task IDs can simplify coNCePTuaL programs that might otherwise need to evaluate complex expressions to determine peer tasks. coNCePTuaL enables either a specific or a randomly selected task to be assigned to a given processor:

⟨*processor_stmt*⟩ ::= ⟨*source_task*⟩
`IS ASSIGNED TO`
`PROCESSOR` ⟨*expr*⟩ `| A RANDOM PROCESSOR`

In addition to performing the specified processor assignment, coNCePTuaL will perform an additional, implicit processor assignment in order to maintain a bijection between task IDs and processor IDs (i.e., every task ID corresponds to exactly one processor ID and every processor ID corresponds to exactly one task ID). Consider the following statement:

```
TASK n SUCH THAT n<(num_tasks+1)/2 IS ASSIGNED TO PROCESSOR n*2
```

If `num_tasks` is '8' the preceding statement will cause '`TASK 0`' to refer to processor 0, '`TASK 1`' to refer to processor processor 2, '`TASK 2`' to refer to processor 4, and '`TASK 3`' to refer to processor 6. What may be unintuitive is that the remaining tasks will not map to their original processors, as doing so would violate the bijection invariant. To clarify coNCePTuaL's implicit processor assignments the following timeline illustrates the execution of '`TASK n SUCH THAT n<(num_tasks+1)/2 IS ASSIGNED TO PROCESSOR n*2`' when `num_tasks` is '8':

'n'	'TASK 0'	'TASK 1'	'TASK 2'	'TASK 3'	'TASK 4'	'TASK 5'	'TASK 6'	'TASK 7'
—	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	0	**2**	**1**	3	4	5	6	7
2	0	2	**4**	3	**1**	5	6	7
3	0	2	4	**6**	1	5	**3**	7

Initially, task and processor IDs are equal. When `n` takes on the value 0, coNCePTuaL performs the equivalent of '`TASK 0 IS ASSIGNED TO PROCESSOR 0`', which does not change the task ID to processor ID mapping. When `n` is 1, coNCePTuaL performs the equivalent of '`TASK 1 IS ASSIGNED TO PROCESSOR 2`', which sets task 1's processor to 2. However, because task 2 also has processor 2, coNCePTuaL implicitly performs the equivalent of '`TASK 2 IS ASSIGNED TO PROCESSOR 1`' in order to preserve the unique task ID to processor ID mapping. When `n` is 2, coNCePTuaL performs the equivalent of '`TASK 2 IS ASSIGNED TO PROCESSOR 4`' and, because task 4 also has processor 4, the equivalent of '`TASK 4 IS ASSIGNED TO PROCESSOR 1`'. Finally, when `n` is 3, coNCePTuaL performs the equivalent of '`TASK 3 IS ASSIGNED TO PROCESSOR 6`' and, because task 6 also has processor 6, the equivalent of '`TASK 6 IS ASSIGNED TO PROCESSOR 3`'. Thus, coNCePTuaL maintains the invariant that after any processor assignment every task corresponds to a unique processor and every processor corresponds to a unique task.

4.6.6 Injecting arbitrary code

There are some features that are outside the scope of the coNCePTuaL language. However, coNCePTuaL provides a mechanism for inserting backend-specific statements

into the control flow of a coNCePTuaL program. This feature is intended for users with specific needs that can't be satisfied through the conventional coNCePTuaL statements.

⟨*backend_stmt*⟩ ::= ⟨*source_task*⟩
BACKEND EXECUTES
⟨*expr*⟩ | ⟨*string*⟩
[AND ⟨*expr*⟩ | ⟨*string*⟩]*

The following example assumes a C-based backend:

```
ALL TASKS taskID BACKEND EXECUTE "my_c_function(" AND taskID AND ");"
```

The `my_c_function()` function needs be defined in some object file and linked with the coNCePTuaL-generated code.

Most users will never need to `BACKEND EXECUTE` code. In fact, most users should *not* use ⟨*backend_stmt*⟩s as they produce nonportable code. One of coNCePTuaL's goals is for programs to be understandable by people unfamiliar with the language, and ⟨*backend_stmt*⟩s thwart that goal. However, ⟨*backend_stmt*⟩s do help ensure that all of the target language/library's features are available to coNCePTuaL.

4.7 Complex statements

The coNCePTuaL statements presented in Section 4.4 [Communication statements], page 63, Section 4.5 [I/O statements], page 70, and Section 4.6 [Other statements], page 73 are all known as *simple statements*. This section expands upon the statements already introduced by presenting *complex statements*. In its most basic form, a ⟨*complex_stmt*⟩ is just a ⟨*simple_stmt*⟩. However, the primary purpose of a ⟨*complex_stmt*⟩ is to juxtapose simple statements and other complex statements into more expressive forms.

Complex statements take one of the following forms, each of which is explained later in this section:

⟨*complex_stmt*⟩ ::= ⟨*simple_stmt*⟩ [THEN ⟨*complex_stmt*⟩]

⟨*simple_stmt*⟩ ::= FOR ⟨*expr*⟩ REPETITIONS [PLUS ⟨*expr*⟩ WARMUP REPETITIONS [AND A SYNCHRONIZATION]] ⟨*simple_stmt*⟩
| FOR EACH ⟨*ident*⟩ IN ⟨*range*⟩ [',' ⟨*range*⟩]* ⟨*simple_stmt*⟩
| FOR ⟨*expr*⟩ ⟨*time_unit*⟩ ⟨*simple_stmt*⟩
| LET ⟨*let_binding*⟩ [AND ⟨*let_binding*⟩]* WHILE ⟨*simple_stmt*⟩
| IF ⟨*rel_expr*⟩ THEN ⟨*simple_stmt*⟩ [OTHERWISE ⟨*simple_stmt*⟩]

In addition, this section explains how to group multiple ⟨*complex_stmt*⟩s into a single semantic unit.

4.7.1 Combining statements

The `THEN` keyword separates statements that are to be performed sequentially. For example, a simple ping-pong communication can be expressed as follows:

```
ALL TASKS RESET ALL COUNTERS THEN
TASK 0 SENDS A 0 BYTE MESSAGE TO TASK 1 THEN
TASK 1 SENDS A 0 BYTE MESSAGE TO TASK 0 THEN
TASK 0 LOGS elapsed_usecs/2 AS "One-way latency"
```

There is no implicit intertask synchronization across THEN statements. Consequently, the two communications specified in the following statement will be performed concurrently:

```
TASK 0 ASYNCHRONOUSLY SENDS AN 8 KILOBYTE MESSAGE TO TASK 1 THEN
TASK 1 ASYNCHRONOUSLY SENDS AN 8 KILOBYTE MESSAGE TO TASK 0 THEN
ALL TASKS AWAIT COMPLETION
```

4.7.2 Iterating

coNCePTuaL provides a variety of looping constructs designed to repeatedly execute a ⟨*simple_stmt*⟩.

Counted loops

The simplest form of iteration in coNCePTuaL repeats a ⟨*simple_stmt*⟩ a given number of times. The syntax is simply "FOR ⟨*expr*⟩ REPETITIONS ⟨*simple_stmt*⟩". As could be expected, the ⟨*expr*⟩ term (see Section 4.2.1 [Arithmetic expressions], page 48) specifies the number of repetitions to perform. Hence, the following ⟨*simple_stmt*⟩ outputs the phrase "I will not talk in class" 100 times:

```
FOR 100 REPETITIONS ALL TASKS OUTPUT "I will not talk in class."
```

FOR. . .REPETITIONS can optionally specify a number of "warmup" repetitions to perform in addition to the base number of repetitions. The syntax is "FOR ⟨*expr*⟩ REPETITIONS PLUS ⟨*expr*⟩ WARMUP REPETITIONS ⟨*simple_stmt*⟩". During warmup repetitions, the OUTPUTS statement (see Section 4.5.2 [Writing to standard output], page 71), the LOGS statement (see Section 4.5.3 [Writing to a log file], page 71), and the COMPUTES AGGREGATES statement (see Section 4.5.3 [Writing to a log file], page 71) are all suppressed (i.e., they have no effect). Many benchmarks synchronize all tasks after performing a set of warmup repetitions. This behavior can be expressed conveniently as part of a coNCePTuaL FOR loop by appending the AND A SYNCHRONIZATION clause:

```
FOR 1000 REPETITIONS PLUS 3 WARMUP REPETITIONS AND A SYNCHRONIZATION
TASK 0 MULTICASTS A 1 MEGABYTE MESSAGE TO ALL OTHER TASKS
```

coNCePTuaL also provides a separate SYNCHRONIZES statement. This is described in Section 4.4.6 [Synchronizing], page 70.

The importance of performing warmup repetitions is that many communication layers give atypically poor performance on the first few transmissions. This may be because the messages miss in the cache; because the communication layer needs to establish connections between pairs of communicating tasks; or, because the operating system needs to "register" message buffers with the network interface. Regardless of the reason, specifying warmup repetitions helps make performance measurements less variable.

Range loops

Range loops are coNCePTuaL's most powerful looping construct. Unlike the counted-loop construct presented above, a range loop binds a variable to a different value on each iteration. Range loops have the following syntax:

FOR EACH ⟨*ident*⟩
IN ⟨*range*⟩ [, ⟨*range*⟩]*

⟨*simple_stmt*⟩

A ⟨*range*⟩ is a comma-separated list of ⟨*expr*⟩s within curly braces. An ellipsis can be used to indicate that coNCePTuaL should fill in the missing numbers. More formally,

⟨*range*⟩ ::= '{'
⟨*expr*⟩ ['**,**' ⟨*expr*⟩]*
['**, ... ,**' ⟨*expr*⟩]
'**}**'

For a range loop, coNCePTuaL successively binds a ⟨*restricted_ident*⟩ (see Section 4.3.1 [Restricted identifiers], page 61) to each ⟨*expr*⟩ in each ⟨*range*⟩ then evaluates the given ⟨*simple_stmt*⟩. Unlike all other uses of a ⟨*restricted_ident*⟩, the variable is not implicitly constrained to being a number from 0 to *numtasks*-1 within a `FOR EACH` statement.

The following are some examples of the `FOR EACH` statement from simplest to most elaborate. Each example uses '`i`' as the loop variable and '`TASK 0 OUTPUTS i`' as the loop body. The output from each example is shown with a "⊣" symbol preceding each line.

```
FOR EACH i IN {8, 7, 5, 4, 5} TASK 0 OUTPUTS i
⊣ 8
⊣ 7
⊣ 5
⊣ 4
⊣ 5

FOR EACH i IN {1, ..., 5} TASK 0 OUTPUTS i
⊣ 1
⊣ 2
⊣ 3
⊣ 4
⊣ 5

FOR EACH i IN {5, ..., 1} TASK 0 OUTPUTS i
⊣ 5
⊣ 4
⊣ 3
⊣ 2
⊣ 1

FOR EACH i IN {1, ..., 5}, {8, 7, 5, 4, 5} TASK 0 OUTPUTS i
⊣ 1
⊣ 2
⊣ 3
⊣ 4
⊣ 5
⊣ 8
⊣ 7
⊣ 5
⊣ 4
⊣ 5
```

```
FOR EACH i IN {1, 4, 7, ..., 30} TASK 0 OUTPUTS i
 ⊣ 1
 ⊣ 4
 ⊣ 7
 ⊣ 10
 ⊣ 13
 ⊣ 16
 ⊣ 19
 ⊣ 22
 ⊣ 25
 ⊣ 28

FOR EACH i IN {3**1, 3**2, 3**3, ..., 3**7} TASK 0 OUTPUTS i
 ⊣ 3
 ⊣ 9
 ⊣ 27
 ⊣ 81
 ⊣ 243
 ⊣ 729
 ⊣ 2187

FOR EACH i IN {0}, {1, 2, 4, ..., 256} TASK 0 OUTPUTS i
 ⊣ 0
 ⊣ 1
 ⊣ 2
 ⊣ 4
 ⊣ 8
 ⊣ 16
 ⊣ 32
 ⊣ 64
 ⊣ 128
 ⊣ 256
```

When coNCePTuaL fills in the numbers summarized by '`...`' it first looks for an arithmetic progression, then a geometric progression. If coNCePTuaL finds neither an arithmetic nor a geometric progression, it re-evaluates all of the ⟨*expr*⟩s in floating-point context (see Section 4.2.1 [Arithmetic expressions], page 48) and tries once again to find a geometric progression. If a pattern is still not found, the compiler aborts with a run-time error message. This is why the final example above places the '`0`' in a separate range; '`1, 2, 4, ..., 256`' is a geometric pattern in which each number is twice the previous. '`0, 1, 2, 4, ..., 256`' violates that pattern, because 1 is not twice 0. The number following the ellipsis is not included when calculating the pattern. Hence, the '`{1, 4, 7, ..., 30}`' range shown above is acceptable to coNCePTuaL. If the number following the ellipsis is less than (respectively, greater than) the first number in an increasing (respectively, decreasing) range, then the loop will silently not execute. Here are some examples to help make those points:

```
FOR EACH i IN {20, 30, 40, ..., 55} TASK 0 OUTPUTS i
 ⊣ 20
 ⊣ 30
```

```
 ⊣ 40
 ⊣ 50

FOR EACH i IN {20, 30, 40, ..., 30} TASK 0 OUTPUTS i
 ⊣ 20
 ⊣ 30

FOR EACH i IN {20, 30, 40, ..., 20} TASK 0 OUTPUTS i
 ⊣ 20

FOR EACH i IN {20, 30, 40, ..., 10} TASK 0 OUTPUTS i
 ⊣ [no output]
```

The examples so far have all used constant expressions. However, as indicated by the definition of ⟨*range*⟩ above, any arbitrary arithmetic expression (see Section 4.2.1 [Arithmetic expressions], page 48) is valid. Assuming that 'x' is currently bound to 123, the following example will produce the indicated output:

```
FOR EACH i IN {x*10, x*9, x*8, ..., x*3}, {x+50}, {2*x+3, 3*x+2}
TASK 0 OUTPUTS i
 ⊣ 1230
 ⊣ 1107
 ⊣ 984
 ⊣ 861
 ⊣ 738
 ⊣ 615
 ⊣ 492
 ⊣ 369
 ⊣ 173
 ⊣ 249
 ⊣ 371
```

Finally, `FOR EACH` loops with constant progressions are executed exactly once. Note that if the ⟨*range*⟩ does not contain an ellipsis then all values are used, regardless of order or constancy:

```
FOR EACH i IN {4, 4, 4, ..., 4} TASK 0 OUTPUTS i
 ⊣ 4

FOR EACH i IN {4, 4, 4, 4, 4} TASK 0 OUTPUTS i
 ⊣ 4
 ⊣ 4
 ⊣ 4
 ⊣ 4
 ⊣ 4
```

Timed loops

A *timed loop* is similar to a counted loop but instead of running for a given number of iterations it runs for a given length of time. Timed loops are absent from all general-purpose programming languages but can be quite useful in the context of network correctness and performance testing. The syntax of coNCePTuaL's timed-loop construct is "`FOR` ⟨*expr*⟩

⟨*time_unit*⟩ ⟨*simple_stmt*⟩". ⟨*time_unit*⟩ is unit of time as listed in Section 4.6.3 [Delaying execution], page 74 and ⟨*expr*⟩ specified the number of ⟨*time_unit*⟩s for which to execute.

The following example shows how to spend three seconds sending messages from task 0 to task 1:

```
FOR 3 SECONDS TASK 0 SENDS A 1 MEGABYTE MESSAGE TO TASK 1
```

Although coNCePTuaL tries its best to run for exactly the specified length of time there will invariably be some error in the process. Always use '`elapsed_usecs`' (see Section A.2 [Predeclared variables], page 127) as the indicator of actual time instead of the time requested in the loop.

4.7.3 Binding variables

There are four ways to bind a value to a variable:

1. as part a source or target task description (see Section 4.3 [Task descriptions], page 61)
2. as part of a range loop (see Section 4.7.2 [Iterating], page 79)
3. via a command-line argument (see Section 4.8.2 [Command-line arguments], page 86)
4. explicitly using the `LET` keyword (this section)

The `LET` statement has the following form:

```
LET ⟨let_binding⟩
[AND ⟨let_binding⟩]*
WHILE ⟨simple_stmt⟩
```

where ⟨*let_binding*⟩ is defined as follows:

```
⟨let_binding⟩ ::= ⟨ident⟩
                  BE
                  ⟨expr⟩ | A RANDOM TASK [      OTHER THAN ⟨expr⟩
                                         |   LESS THAN ⟨expr⟩ [BUT NOT ⟨expr⟩]
                                         |   GREATER THAN ⟨expr⟩ [BUT NOT ⟨expr⟩]]
```

Here are some examples of `LET`:

```
LET reps BE 3 WHILE FOR reps REPETITIONS TASK 0 OUTPUTS "Laissez les
bons temps rouler."

LET src BE numtasks-1 AND dest BE numtasks/2 WHILE TASK src SENDS A
55E6 BIT MESSAGE TO TASK dst

LET hotspot BE A RANDOM TASK WHILE TASKS other SUCH THAT
other<>hotspot SEND 1000 1 MEGABYTE MESSAGES TO TASK hotspot

LET target BE A RANDOM TASK OTHER THAN 3 WHILE TASK 3 SENDS A 24 BYTE
MESSAGE TO TASK target

LET x BE A RANDOM TASK AND y be a RANDOM TASK GREATER THAN x WHILE
TASK 0 OUTPUTS "Did you know that " AND x AND " is less than " AND y
AND "?"

LET nonzero BE A RANDOM TASK LESS THAN 10 BUT NOT 0 WHILE TASK nonzero
```

```
SLEEPS FOR 2 SECONDS

LET num BE 1000 AND num BE num*2 WHILE TASK 0 OUTPUTS num
⊣ 2000
```

The last example demonstrates that `LET` can bind a variable to a function of its previous value. It is important to remember, though, that variables in coNCePTuaL cannot be assigned, only bound to a value for the duration of the current scope. They can, however, be bound to a different value for the duration of a child scope. The following example is an attempt to clarify the distinction between binding and assignment:

```
LET var BE 123 WHILE FOR 5 REPETITIONS LET var BE var+1 WHILE TASK 0
OUTPUTS var
⊣ 124
⊣ 124
⊣ 124
⊣ 124
⊣ 124
```

In that example, '`var`' is bound to '`123`' for the scope containing the `FOR` statement. Then, within the `FOR` statement, a new scope begins with '`var`' being given one plus the value it had in the outer scope, resulting in '`124`'. If coNCePTuaL supported assignment instead of variable-binding, the program would have output '`124`', '`125`', '`126`', '`127`', and '`128`'. Note that if `A RANDOM TASK` were used in the example instead of '`var+1`', '`var`' would get a different value in each iteration.

When a variable is `LET`-bound to `A RANDOM TASK`, all tasks agree on the random number. Otherwise, task A might send a message to task B but task B might be expecting to receive the message from task C, thereby leading to a variety of problems. If there are no valid random tasks, as in the following example, `A RANDOM TASK` will return '`-1`':

```
LET invalid_task BE A RANDOM TASK GREATER THAN num_tasks WHILE TASK 0
OUTPUTS invalid_task
⊣ -1
```

Furthermore, the ⟨*expr*⟩ passed to `GREATER THAN` is bounded from below by '`0`' and the ⟨*expr*⟩ passed to `LESS THAN` is bounded from above by '`num_tasks-1`'. Hence, the following coNCePTuaL statement will always output values less than or equal to '`num_tasks-1`' (unless `num_tasks` is greater than 1×10^6, of course, in which case it will always output values less than 1×10^6):

```
LET valid_task BE A RANDOM TASK LESS THAN 1E6 WHILE TASK valid_task
OUTPUTS "Hello from " AND valid_task
```

4.7.4 Conditional execution

Like most programming languages, coNCePTuaL supports conditional code execution:

⟨*if_stmt*⟩ ::= `IF` ⟨*rel_expr*⟩
`THEN` ⟨*simple_stmt*⟩
[`OTHERWISE` ⟨*simple_stmt*⟩]

The semantics of an ⟨*if_stmt*⟩ are that if ⟨*rel_expr*⟩ (see Section 4.2.5 [Relational expressions], page 60) is TRUE then the first ⟨*simple_stmt*⟩ is executed. If ⟨*rel_expr*⟩ is FALSE then the second ⟨*simple_stmt*⟩ is executed. One restriction is that ⟨*rel_expr*⟩ must return the

same truth value to every task. Consequently, functions that involve task-specific random numbers (see Section 4.2.2 [Built-in functions], page 49) are forbidden within ⟨*rel_expr*⟩.

The following is an example of an ⟨*if_stmt*⟩:

```
IF this>that THEN TASK 0 SENDS A 3 KILOBYTE MESSAGE TO TASK this
OTHERWISE TASK num_tasks-1 SENDS A 4 KILOBYTE MESSAGE TO TASK that
```

4.7.5 Grouping

`FOR` loops, `LET` bindings, and `IF` statements operate on a single ⟨*simple_stmt*⟩ (or two ⟨*simple_stmt*⟩s in the case of `IF...OTHERWISE`). Operating on multiple ⟨*simple_stmt*⟩s—or, more precisely, operating on a single ⟨*complex_stmt*⟩ that may consist of multiple ⟨*simple_stmt*⟩s—is a simple matter of placing the ⟨*simple_stmt*⟩s within curly braces. Contrast the following:

```
FOR 3 REPETITIONS TASK 0 OUTPUTS "She loves me." THEN TASK 0 OUTPUTS
"She loves me not."
⊣ She loves me.
⊣ She loves me.
⊣ She loves me.
⊣ She loves me not.

FOR 3 REPETITIONS {TASK 0 OUTPUTS "She loves me." THEN TASK 0 OUTPUTS
"She loves me not."}
⊣ She loves me.
⊣ She loves me not.
⊣ She loves me.
⊣ She loves me not.
⊣ She loves me.
⊣ She loves me not.
```

In other words, everything between '`{`' and '`}`' is treated as if it were a single statement. Hence, the `FOR` loop applies only to the "She loves me" output in the first statement above, while the `FOR` loop applies to both "She loves me" and "She loves me not" in the second statement.

Variable scoping is limited to the ⟨*simple_stmt*⟩ in the body of a `LET`:

```
LET year BE 1984 WHILE LET year BE 2084 WHILE TASK 0 OUTPUTS year THEN
TASK 0 OUTPUTS year
```

error The second '`year`' is outside the scope of both '`LET`' statements.

```
LET year BE 1984 WHILE {{LET year BE 2084 WHILE TASK 0 OUTPUTS year}
THEN TASK 0 OUTPUTS year}
⊣ 2084
⊣ 1984
```

4.8 Header declarations

coNCePTuaL programs may contain a header section that precedes the first statement in a coNCePTuaL program. The header section contains two types of declarations that affect the remainder of the program: language versioning declarations and declarations of command-line options.

4.8.1 Language versioning

Because the coNCePTuaL language is still under development, the programmer is forewarned that major changes are likely. Changes may prevent old code from compiling or, even worse, may cause old code to produce incorrect results (e.g., if scoping or block structuring are altered). To mitigate future language changes coNCePTuaL enables programs to specify which version of the language they were written to. The syntax is straightforward:

⟨*version_decl*⟩ ::= `REQUIRE LANGUAGE VERSION` ⟨*string*⟩

The parser issues a warning message if ⟨*string*⟩ does not exactly match the language version supported by the compiler. If the program successfully compiles after a version-mismatch warning, the programmer should check the output very carefully to ensure that the program behaved as expected.

The current version of the coNCePTuaL language is '`0.5.2a`'. Note that the language version does not necessarily correspond to the version of the coNCePTuaL toolset (see Chapter 3 [Usage], page 11) as a whole.

4.8.2 Command-line arguments

coNCePTuaL makes it easy to declare command-line parameters, although the syntax is a bit verbose:

⟨*param_decl*⟩ ::= ⟨*ident*⟩
`IS` ⟨*string*⟩
`AND COMES FROM` ⟨*string*⟩ `OR` ⟨*string*⟩
`WITH DEFAULT` ⟨*number*⟩

⟨*ident*⟩ is the coNCePTuaL variable being declared. The first ⟨*string*⟩ is a descriptive string that is provided when the user runs the program with `--help` or `-h`. The '⟨*string*⟩ `OR` ⟨*string*⟩' terms list the long name for the command-line option, preceded by '`--`', and the short (single-character) name, preceded by '`-`'. Finally, ⟨*number*⟩ specifies the value that will be assigned to ⟨*ident*⟩ if the command-line option is not used. Note that short names (also long names) must be unique.

For instance, the declaration '`nummsgs IS "Number of messages to send" AND COMES FROM "--messages" OR "-m" WITH DEFAULT 100`' declares a new coNCePTuaL variable called '`nummsgs`'. '`nummsgs`' is given the value '`100`' by default. However, if the user running the program specifies, for example, `--messages=55` (or, equivalently, `-m 55`), then '`nummsgs`' will be given the value '`55`'. The following is an example of the output that might be produced if the program is run with `--help` or `-h`:

```
Usage: a.out [OPTION...]
  -m, --messages=<number>     Number of messages to send [default: 100]
  -C, --comment=<string>      Additional commentary to write to the log
                              file, @FILE to import commentary from FILE,
                              or !COMMAND to import commentary from COMMAND
                              (may be specified repeatedly)
  -L, --logfile=<string>      Log file template [default: "a.out-%p.log"]
  -N, --no-trap=<string>      List of signals that should not be trapped
                              [default: ""]
  -W, --watchdog=<number>     Number of minutes after which to kill the job
                              (-1=never) [default: -1]

Help options:
  -?, --help                  Show this help message
  --usage                     Display brief usage message
```

The above is only an example. Depending on what libraries were available when the coNCePTuaL run-time system was configured, the output could be somewhat different. Also, long options may not be supported if a suitable argument-processing library was not available at configuration time. The above example does indicate one way that help strings could be formatted. It also shows that the coNCePTuaL run-time system reserves some command-line options for its own purposes. Currently, these all use uppercase letters for their short forms so it should be safe for programs to use any lowercase letter.

4.9 Complete programs

A complete coNCePTuaL program consists of zero or more header declarations (see Section 4.8 [Header declarations], page 85), each terminated with a '.', followed by one or more complex statements (see Section 4.7 [Complex statements], page 78), each also terminated with a '.'. More formally, coNCePTuaL's top-level nonterminal is the ⟨*program*⟩:

⟨*program*⟩ ::= (⟨*version_decl*⟩ | ⟨*param_decl*⟩ ['.'])*
(⟨*top_level_complex_stmt*⟩ ['.'])+

in which a ⟨*top_level_complex_stmt*⟩ reduces immediately to a ⟨*complex_stmt*⟩. Because a ⟨*complex_stmt*⟩ can reduce to a ⟨*simple_stmt*⟩, the most basic, complete coNCePTuaL program would be a ⟨*simple_stmt*⟩ with a terminating period:

```
ALL TASKS self OUTPUT "Hello from task " AND self AND "!".
```

A fuller example might contain multiple header declarations and multiple ⟨*complex_stmt*⟩s:

```
# A complete coNCePTuaL program
# By Scott Pakin <pakin@lanl.gov>

REQUIRE LANGUAGE VERSION "0.5.2a".

maxval IS "Maximum value to loop to" AND COMES FROM "--maxval" OR
  "-v" WITH DEFAULT 100.

step IS "Increment after each iteration" AND COMES FROM "--step" OR
  "-s" WITH DEFAULT 1.
```

```
TASK 0 OUTPUTS "Looping from 0 to " AND maxval AND " by " AND step
  AND "...".

FOR EACH loopvar IN {0, step, ..., maxval}
  TASK 0 OUTPUTS "    " AND loopvar.

TASK 0 OUTPUTS "Wasn't that fun?".
```

Technically, the '`.`' is optional; the language is unambiguous without it. However, for aesthetic purposes it is recommended that you terminate sentences with a period, just like in a natural language. An exception would be when a ⟨*complex_stmt*⟩ ends with a curly brace. The '`}.`' syntax is unappealing so a simple '`}`' should be used instead. See Chapter 5 [Examples], page 89, for further examples.

Top-level statements and log files

The reason that coNCePTuaL distinguishes between ⟨*top_level_complex_stmt*⟩s and ⟨*complex_stmt*⟩s is that ⟨*top_level_complex_stmt*⟩s begin a new table in the log file (see Section 4.5.3 [Writing to a log file], page 71) while ⟨*complex_stmt*⟩s add columns to the current table. Consider the following piece of code:

```
TASK 0 LOGS 111 AS "First" AND
            222 AS "Second".
```

Because '`First`' and '`Second`' are logged within the same ⟨*simple_stmt*⟩ they appear in the log file within the same table but as separate columns:

```
"First","Second"
"(all data)","(all data)"
111,222
```

The same rule holds when `LOGS` is used repeatedly across ⟨*simple_stmt*⟩s but within the same ⟨*complex_stmt*⟩:

```
TASK 0 LOGS 111 AS "First" THEN
TASK 0 LOGS 222 AS "Second".
```

However, if '`First`' and '`Second`' are logged from separate ⟨*top_level_complex_stmt*⟩s, the coNCePTuaL run-time library stores them in separate tables:

```
TASK 0 LOGS 111 AS "First".
TASK 0 LOGS 222 AS "Second".
```

```
"First"
"(all data)"
111

"Second"
"(all data)"
222
```

5 Examples

This chapter presents a variety of examples of complete coNCePTuaL programs. The purpose is to put in context the grammatical elements described in Chapter 4 [Grammar], page 46 and also to illustrate coNCePTuaL's power and expressiveness.

5.1 Latency

One of the most common network performance benchmarks is a ping-pong latency test. Not surprisingly, such a test is straightforward to implement in coNCePTuaL:

```
# A ping-pong latency test written in coNCePTuaL

Require language version "0.5.2a".

# Parse the command line.
reps is "Number of repetitions of each message size" and comes from
 "--reps" or "-r" with default 1000.
maxbytes is "Maximum number of bytes to transmit" and comes from
 "--maxbytes" or "-m" with default 1M.

# Ensure the we have a peer with whom to communicate.
Assert that "the latency test requires at least two tasks" with
  num_tasks>=2.

# Perform the benchmark.
For each msgsize in {0}, {1, 2, 4, ..., maxbytes} {
  for reps repetitions {
    task 0 resets its counters then
    task 0 sends a msgsize byte message to task 1 then
    task 1 sends a msgsize byte message to task 0 then
    task 0 logs the msgsize as "Bytes" and
                the median of elapsed_usecs/2 as "1/2 RTT (usecs)"
  } then
  task 0 computes aggregates
}
```

Note that the outer `FOR` loop specifies two ⟨*range*⟩s (see Section 4.7.2 [Iterating], page 79). This is because '`{0, 1, 2, 4, ..., maxbytes}`' is not a geometric progression. Hence, that incorrect ⟨*range*⟩ is split into the singleton '`{0}`' and the geometric progression '`{1, 2, 4, ..., maxbytes}`'.

5.2 Hot potato

One way to measure performance variance in a cluster is with a "hot potato" test. The idea is that the tasks send a message in a ring pattern, then the first task logs the minimum, mean, and variance of the per-hop latency. Ideally, the minimum should equal the mean and these should both maintain a constant value as the number of tasks increases. Also,

the variance should be small and constant. The following coNCePTuaL code implements a hot-potato test.

```
# Virtual ring "hot potato" test

Require language version "0.5.2a".

trials is "Number of trials to perform" and comes from "--trials" or
  "-t" with default 100000.

Assert that "the hot-potato test requires at least two tasks" with
  num_tasks>=2.

Let len be 0 while {
  for each task_count in {2, ..., num_tasks} {
    task 0 outputs "Performing " and trials and " " and
                   task_count and "-task runs...." then
    for trials repetitions plus 5 warmup repetitions {
      task 0 resets its counters then
      task 0 sends a len byte message to unsuspecting task 1 then
      task (n+1) mod task_count receives a len byte message from task
        n such that n<task_count then
      task n such that n>0 /\ n<task_count sends a len byte message
        to unsuspecting task (n+1) mod task_count then
      task 0 logs the task_count as "# of tasks" and
                  the minimum of elapsed_usecs/task_count as
                    "Latency (usecs)" and
                  the mean of elapsed_usecs/task_count as
                    "Latency (usecs)" and
                  the variance of elapsed_usecs/task_count as
                    "Latency (usecs)"
    } then
    task 0 computes aggregates
  }
}
```

All tasks receive from their left neighbor and send to their right neighbor. However, in order to avoid a deadlock situation, task 0 sends then receives while all of the other tasks receive then send.

5.3 Hot spot

Different systems react differently to resource contention. A hot-spot test attempts to measure the performance degradation that occurs when a task is flooded with data. That is, all tasks except 0 concurrently send a batch of messages to task 0. Task 0 reports the incoming bandwidth, i.e., the number of bytes it received divided by the time it took to receive that many bytes. The two independent variables are the message size and the number of tasks.

```
# Hot-spot bandwidth

Require language version "0.5.2a".

maxbytes is "Maximum message size in bytes" and comes from
  "--maxbytes" or "-x" with default 1024.
numtrials is "Number of bursts of each size" and comes from "--trials"
  or "-t" with default 100.
burst is "Number of messages in each burst" and comes from
  "--burstsize" or "-b" with default 1000.

Assert that "the hot-spot test requires at least two tasks" with
  num_tasks>=2.

For each maxtask in {2, ..., num_tasks}
  for each msgsize in {1, 2, 4, ..., maxbytes} {
    task 0 outputs "Performing " and numtrials and " " and
                   maxtask and "-task trials with " and
                   msgsize and "-byte messages" then
    for numtrials repetitions plus 3 warmup repetitions {
      task 0 resets its counters then
      task sender such that sender>0 /\ sender<maxtask asynchronously
        sends burst msgsize byte messages to task 0 then
      all tasks await completion then
      task 0 logs the maxtask as "Tasks" and
                  the msgsize as "Message size (B)" and
                  the mean of (1E6*bytes_received)/(1M*elapsed_usecs)
                    as "Incoming BW (MB/s)"
    } then
    task 0 computes aggregates
  }
```

5.4 Multicast trees

It may be worth comparing the performance of a native multicast operation to the performance achieved by multicasting over a k-nomial tree to gauge how well the underlying communication layer implements multicasts. The following code records a wealth of data, varying the tree arity (i.e., k), the number of tasks receiving the multicast, and the message size. It provides a good demonstration of how to use the `KNOMIAL_CHILDREN` and `KNOMIAL_CHILD` functions.

```
# Test the performance of multicasting over various k-nomial trees
# By Scott Pakin <pakin@lanl.gov>

Require language version "0.5.2a".

# Parse the command line.
minsize is "Min. message size (bytes)" and comes from "--minbytes" or "-n"
  with default 1.
maxsize is "Max. message size (bytes)" and comes from "--maxbytes" or "-x"
  with default 1M.
reps is "Repetitions to perform" and comes from "--reps" or "-r" with
  default 100.
maxarity is "Max. arity of the tree" and comes from "--maxarity" or "-a"
  with default 2.

Assert that "this program requires at least two processors" with
  num_tasks>=2.

# Send messages from task 0 to 1, 2, 3, ... other tasks in a k-nomial tree.
For each arity in {2, ..., maxarity} {
  for each num_targets in {1, ..., num_tasks-1} {
    for each msgsize in {minsize, minsize*2, minsize*4, ..., maxsize} {
      task 0 outputs "Multicasting a " and msgsize and "-byte message to "
        and num_targets and " target(s) over a " and arity and
        "-nomial tree ..." then
      for reps repetitions {
        task 0 resets its counters then
        for each src in {0, ..., num_tasks}
          for each dstnum in {0, ..., knomial_children(src, arity,
              num_targets+1)}
            task src sends a msgsize byte message to task
              knomial_child(src, dstnum, arity) then
        all tasks synchronize then
        task 0 logs the arity as "k-nomial arity" and
                    the num_targets as "# of recipients" and
                    the msgsize as "Message size (bytes)" and
                    the median of (1E6/1M)*(msgsize/elapsed_usecs) as
                      "Incoming bandwidth (MB/s)" and
                    the median of (num_targets*msgsize/elapsed_usecs)*
                      (1E6/1M) as "Outgoing bandwidth (MB/s)"
      } then
      task 0 computes aggregates
    }
  }
}
```

6 Implementation

coNCePTuaL could have been implemented as a benchmarking library instead of as a special-purpose language. In addition to improved readability and the practicality of including entire source programs in every log file, one advantage of the language approach is that the same coNCePTuaL source code can be used to compare the performance of multiple communication libraries. A compiler command-line option selects a particular backend module to use to generate code. Each backend outputs code for a different combination of low-level language and communication library.

The coNCePTuaL compiler is structured into a pipeline of modules. Thus, the backend can be replaced without altering the front end, lexer, or parser modules. coNCePTuaL ensures consistency across backends by providing a run-time library that generated code can link to. The run-time library encapsulates many of the mundane tasks a network correctness or performance test needs to perform.

6.1 Overview

Compiler

The coNCePTuaL compiler is written in Python and is based on the SPARK (Scanning, Parsing, And Rewriting Kit) compiler framework. Compiler execution follows a basic pipeline structure. Compilation starts with the top-level file (‘`ncptl.py`’), which processes the command line then transfers control to the lexer (‘`ncptl_lexer.py`’). The lexer inputs coNCePTuaL source code and outputs a stream of tokens (‘`token.py`’). Next, the parser (‘`ncptl_parser.py`’) finds structure in those tokens based on coNCePTuaL’s grammatical rules and outputs an abstract syntax tree (‘`ast.py`’). Finally, the code generator (‘`codegen_`*language*`_`*library*`.py`’) that was designated on the command line walks the abstract syntax tree, converting it to code in the target language and for the target communication library.

Run-time library

coNCePTuaL makes a large run-time library (‘`runtimelib.c`’) available to generated programs. The coNCePTuaL run-time library, which is written in C, provides consistent functionality across target languages and communication layers as well as across hardware architectures and operating systems. The library also simplifies code generation by implementing functions for such tasks as memory allocation, queue management, and data logging. The functions in this library are described in Section 6.3 [Run-time library functions], page 102.

Build process

coNCePTuaL is built using the GNU Autotools (Autoconf, Automake, and Libtool). Consequently, changes should be made to original files, not generated files. Specifically, ‘`configure.ac`’ and ‘`acinclude.m4`’ should be edited in place of ‘`configure`’; ‘`ncptl.h.in`’ should be edited in place of ‘`ncptl.h`’; and, the various ‘`Makefile.am`’ files should be edited

in place of the corresponding '`Makefile`'s. See the *Autoconf* documentation, the *Automake* documentation, and the *Libtool* documentation for information about how these various tools operate.

If '`configure`' is given the `--enable-maintainer-mode` option, `make` will automatically re-run `aclocal`, `autoheader`, `automake`, `autoconf`, and/or `./configure` as needed. Developers who plan to modify any of the "maintainer" files ('`acinclude.m4`', '`configure.ac`', and the various '`Makefile.am`' files) are strongly encouraged to configure coNCePTuaL with `--enable-maintainer-mode` in order to ensure that the build process is kept current with any changes.

6.2 Backend creation

The coNCePTuaL compiler's backend generates code from an abstract syntax tree (AST). The compiler was designed to support a variety of code generators, each targeting a particular programming language and communication library. There are two ways to create a new coNCePTuaL backend. Either a '`codegen_`*language*`_`*library*`.py`' backend supporting an arbitrary language and communication library can be written from scratch or a C-based '`codegen_c_`*library*`.py`' backend can be derived from '`codegen_c_generic.py`'.

In the former case, the backend must derive a `NCPTL_CodeGen` class from SPARK's `GenericASTTraversal` class. `NCPTL_CodeGen` class must contain a `generate` method with the following signature:

```
def generate(self, ast, filesource='<stdin>', sourcecode=None):
```

That is, `generate` takes as arguments a class object, the root of an abstract-syntax tree (as defined in '`ast.py`'), the name of the input file containing coNCePTuaL code (to be used for outputting error messages), and the complete coNCePTuaL source code (which is both stored in header comments and passed to the run-time library). `generate` should invoke `self.postorder` to traverse the AST in a post-order fashion, calling various code-generating methods as it proceeds. The `NCPTL_CodeGen` must implement all of the methods listed in Section B.1 [Method calls], page 129, each of which corresponds to some component of the coNCePTuaL grammar. Each method takes a "self" class object an a node of the AST (of type `AST`).

The compiler front-end, '`ncptl`', invokes the following two methods, which must be defined by the backend's `NCPTL_CodeGen` class:

```
def compile_only(self, progfilename, codelines, outfilename,
                 verbose, keepints):
```

```
def compile_and_link(self, progfilename, codelines, outfilename,
                     verbose, keepints):
```

The `compile_only` method compiles the backend-specific code into an object file. The `compile_and_link` method compiles the backend-specific code into an object file and links

it into an executable file. For some backends, the notions of "compile" and "link" may not be appropriate. In that situation, the backend should perform the closest meaningful operations. For example, the `dot` backend (see Section 3.2.6 [The dot backend], page 19) compiles to a '`.dot`' file and links into the target graphics format ('`.ps`' by default).

For both the `compile_only` and `compile_and_link` methods, `progfilename` is the name of the coNCePTuaL input file specified on the '`ncptl`' command line or the string '`<command line>`' if a program was specified literally with `--program`. `codelines` is the output from the `generate` method, i.e., a list of lines of backend-specific code. `outfilename` is the name of the target file specified on the '`ncptl`' command line with `--output` or the string '`-`' if `--output` was not used. If `verbose` is '`1`', the method should write each operation it plans to perform to the standard-error device. For consistency, comment lines should begin with '`#`'; shell commands should be output verbatim. If `verbose` is '`0`', corresponding to the '`ncptl`' `--quiet` option, the method should output nothing but error messages. Finally, `keepints` corresponds to the `--keep-ints` option to '`ncptl`'. If equal to '`0`', all intermediate files should be deleted before returning; if equal to '`1`', intermediate files should be preserved. See Section 3.1 [Compiling coNCePTuaL programs], page 11, for a description of the various command-line options to '`ncptl`'.

As long as `NCPTL_CodeGen` implements all of the required functions it is free to generate code in any way that it sees fit. However, Section B.1 [Method calls], page 129, lists a large number of methods, many of which will be identical across multiple code generators for the same language but different communication libraries. To simplify a common case, C plus some messaging library, coNCePTuaL provides '`codegen_c_generic.py`', to which the remainder of the Implementation chapter is devoted to explaining.

6.2.1 Hook methods

Multiple code generators for the same language but different communication libraries are apt to contain much code in common. Because C is a popular language, coNCePTuaL provides a '`codegen_c_generic.py`' module which implements a virtual `NCPTL_CodeGen` base class. This base class implements all of the methods listed in Section B.1 [Method calls], page 129. However, rather than support a particular communication library, the '`codegen_c_generic.py`' implementation of `NCPTL_CodeGen` merely provides a number of calls to "hook" methods—placeholders that implement library-specific functionality. See Section B.2 [C hooks], page 131, for a list of all of the hooks that '`codegen_c_generic.py`' defines. For clarity, hooks are named after the method from which they're called but with an all-uppercase tag appended. Hook methods take a single parameter, a read-only dictionary (the result of invoking Python's `locals()` function) of all of the local variables in the caller's scope. They return C code in the form of a list with one line of C per element. A hook method is invoked only if it exists, which gives the backend developer some flexibility in selecting places at which to insert code. Of course, for coarser-grained control, a backend developer can override complete methods in '`codegen_c_generic.py`' if desired. Generally, this will not be necessary as hook invocations are scattered liberally throughout the file.

An example

'`codegen_c_generic.py`' defines a method named `code_specify_include_files`. ('`codegen_c_generic.py`' names all of its code-generating helper methods

'`code_`*something*'.) `code_specify_include_files` pushes a sequence of `#include` directives onto a queue of lines of C code. The method is shown below in its entirety:

```
def code_specify_include_files(self, node):
    "Load all of the C header files the generated code may need."

    # Output a section comment.
    self.pushmany([
        "/*****************",
        " * Include files *",
        " *****************/",
        ""])

    # Enable hooks both before and after the common includes.
    self.pushmany(self.invoke_hook("code_specify_include_files_PRE",
                                   locals(),
                                   before=[
        "/* Header files specific to the %s backend */" %
        self.backend_name],
                                   after=[""]))
    self.pushmany([
        "/* Header files needed by all C-based backends */",
        "#include <stdio.h>",
        "#include <string.h>",
        "#include <ncptl/ncptl.h>"])
    self.pushmany(self.invoke_hook("code_specify_include_files_POST",
                                   locals(),
                                   before=[
        "",
        "/* Header files specific to the %s backend */" %
        self.backend_name]))
```

`code_specify_include_files` uses the `pushmany` method (see Section 6.2.4 [Internals], page 100) to push each element in a list of lines of C code onto the output queue. It starts by pushing a section comment—'`codegen_c_generic.py`' outputs fully commented C code. Next, it invokes the `code_specify_include_files_PRE` hook if it exists and pushes that method's return value onto the queue. Then, it pushes all of the `#include`s needed by the generated C code. Finally, it invokes the `code_specify_include_files_POST` hook if it exists and pushes that method's return value onto the queue.

A backend that requires additional header files from those included by `code_specify_include_files` need only define `code_specify_include_files_PRE` to add extra header files before the standard ones or `code_specify_include_files_POST` to add extra header files after them. The following is a sample (hypothetical) hook definition:

```
def code_specify_include_files_POST(self, localvars):
    "Specify extra header files needed by the c_pthreads backend."
    return [
        "#include <errno.h>",
        "#include <pthread.h>"]
```

Although the top-level structure of 'codegen_c_generic.py' is described in Section 6.2.4 [Internals], page 100, a backend developer will normally need to study the 'codegen_c_generic.py' source code to discern the purpose of each hook method and its relation to the surrounding code.

6.2.2 A minimal C-based backend

A backend derived from 'codegen_c_generic.py' starts by defining an NCPTL_CodeGen child class that inherits much of its functionality from the parent NCPTL_CodeGen class. There are only two items that a C-based backend *must* define: backend_name, the name of the backend in the form '*c_library*'; and, backend_desc, a brief phrase describing the backend. (These are used for error messages and file comments.) Also, a backend's __init__ method must accept an options parameter, which is given a list of command-line parameters not recognized by 'ncptl.py'. After NCPTL_CodeGen's __init__ method processes the entries in options that it recognizes, it should pass the remaining options to its parent class's __init__ method for further processing. (For proper initialization, the parent class's __init__ method must be called, even if there are no remaining options to process.)

The following is the complete source code to a minimal coNCePTuaL backend. This backend, 'codegen_c_seq.py', supports only sequential coNCePTuaL programs (e.g., 'TASK 0 OUTPUTS "Hello, world!"'); any attempt to use communication statements (see Section 4.4 [Communication statements], page 63) will result in a compile-time error.

```
#! /usr/bin/env python

########################################################
# Code generation module for the coNCePTuaL language: #
# Minimal C-based backend -- all communication        #
# operations result in a compiler error               #
#                                                     #
# By Scott Pakin <pakin@lanl.gov>                     #
########################################################

import codegen_c_generic

class NCPTL_CodeGen(codegen_c_generic.NCPTL_CodeGen):
    def __init__(self, options):
        "Initialize the sequential C code generation module."
        self.backend_name = "c_seq"
        self.backend_desc = "C, sequential code only"
        codegen_c_generic.NCPTL_CodeGen.__init__(self, options)
```

The c_seq backend can be used like any other:

```
ncptl --backend=c_seq \
  --program='For each i in {10, ..., 1} task 0 outputs i.' | \
  indent > myprogram.c
```

('`codegen_c_generic.py`' outputs unindented code, deferring attractive formatting to the Unix '`indent`' utility.)

One sequential construct the `c_seq` backend does not support is randomness, as needed by`A RANDOM PROCESSOR` (see Section 4.6.5 [Reordering task IDs], page 76) and `A RANDOM TASK` (see Section 4.7.3 [Binding variables], page 83). '`codegen_c_generic.py`' cannot support randomness itself because doing so requires broadcasting the seed for the random-number generator to all tasks. Broadcasting requires messaging-layer support, which a derived backend provides through the `code_def_init_reseed_BCAST` hook (see Section 6.2.1 [Hook methods], page 95). For the sequential backend presented above, a broadcast can be implemented as a no-op:

```
def code_def_init_reseed_BCAST(self, localvars):
    '"Broadcast" a random-number seed to all tasks.'
    return []
```

In fact, that same do-nothing hook method is used by the `c_udgram` backend. `c_udgram` seeds the random-number generator before calling `fork()`, thereby ensuring that all tasks have the same seed without requiring an explicit broadcast.

6.2.3 Generated code

'`codegen_c_generic.py`' generates thoroughly commented C code. However, the overall structure of the generated code may be somewhat unintuitive, as it does not resemble the code that a human would write to accomplish a similar task. The basic idea behind the generated C code is that it expands the entire program into a list of "events", then starts the clock, then executes all of the events in a single loop. Regardless of the coNCePTuaL program being compiled, the body of the generated C code will look like this:

```
for (i=0; i<numevents; i++) {
  CONC_EVENT *thisev = &eventlist[i];

  switch (thisev->type) {
    case event_1:
                                                  ⋮
    case event_2:
                                                  ⋮
  }
}
```

Programs generated by '`codegen_c_generic.py`' define the following event types:

`EV_SEND` Synchronous send

`EV_ASEND` Asynchronous send

`EV_RECV` Synchronous receive

`EV_ARECV` Asynchronous receive

`EV_WAIT` Wait for all asynchronous sends/receives to complete

`EV_DELAY` Spin or sleep

`EV_TOUCH` Touch a region of memory

`EV_SYNC` Barrier synchronization

`EV_RESET` Reset counters

`EV_FLUSH` Compute aggregate functions for log-file columns

`EV_MCAST` Synchronous multicast

`EV_BTIME` Beginning of a timed loop

`EV_ETIME` Ending of a timed loop

`EV_REPEAT`
Repeatedly process the next N events

`EV_SUPPRESS`
Suppress writing to the log and standard output

`EV_NEWSTMT`
Beginning of a new top-level statement

`EV_CODE` None of the above

The `EV_CODE` event is used, for example, by the `BACKEND EXECUTES` (see Section 4.6.6 [Injecting arbitrary code], page 77), `LOGS` (see Section 4.5.3 [Writing to a log file], page 71), and `OUTPUTS` (see Section 4.5.2 [Writing to standard output], page 71) statements. Note that there are no loop events—in fact, there are no complex statements (see Section 4.7 [Complex statements], page 78) whatsoever. Complex statements are expanded into multiple simple statements at initialization time.

The advantage of completely expanding a coNCePTuaL program during the initialization phase—essentially, "pre-executing" the entire program—is that that enables all of the expensive, non-communication-related setup to be hoisted out of the timing loop, which is how a human would normally express a network benchmark. Pre-execution is possible because the coNCePTuaL language is not a Turing machine; infinite loops are not expressible by the language and message contents and timings cannot affect program behavior, for instance. During its initialization phase, the generated C code allocates memory for message buffers, evaluates numerical expressions, verifies program assertions, unrolls loops, and does everything else that's not directly relevant to communication performance. For instance, the coNCePTuaL program '`TASK tx SUCH THAT tx>4 SENDS 10 1 MEGABYTE MESSAGES TO TASK tx/2`' would cause each task to perform the following steps during initialization:

- determine if its task ID is greater than 4, making the task a sender
- determine if its task ID is equal to '`tx/4`' (rounded down to the nearest integer) for some task '`tx`' in the program, making the task a receiver
- allocate 1 MB for a message buffer

- allocate and initialize a repeat event, specifying that the subsequent event should repeat 10 times
- allocate a send or receive event

The final two of those steps repeat as necessary. For example, task 3 receives 10 messages from each of task 6 and task 7.

Note that each task's receive events (if any) are allocated before its send events (if any), as described Section 4.4.2 [Sending], page 67. Also, note that only a single message buffer is allocated because the coNCePTuaL source did not specify the `UNIQUE` keyword (see Section 4.4.1 [Message specifications], page 63).

An event is implemented as a C `struct` that contains all of the state needed to perform a particular operation. For example, an event corresponding to a synchronous or asynchronous send operation (`CONC_SEND_EVENT`) stores the destination task ID, the number of bytes to send, the message alignment, the number of outstanding asynchronous sends and receives, a flag indicating whether the data is to be touched, and a flag indicating that the message should be filled with data the receiver can verify. In addition, the `code_declare_datatypes_SEND_STATE` hook (see Section 6.2.1 [Hook methods], page 95) enables a backend to include additional, backend-specific state in the (`CONC_SEND_EVENT`) data structure.

6.2.4 Internals

'`codegen_c_generic.py`' is a fairly substantial piece of code. It is divided into ten sections:

1. methods exported to the compiler front end
2. utility functions that do not generate code
3. utility functions that do generate code
4. methods for outputting language atoms (see Section 4.1 [Primitives], page 46)
5. methods for outputting miscellaneous language constructs (e.g., restricted identifiers; see Section 4.3.1 [Restricted identifiers], page 61)
6. methods for outputting expressions (see Section 4.2 [Expressions], page 47)
7. methods for outputting complete programs (see Section 4.9 [Complete programs], page 87)
8. methods for outputting complex statements (see Section 4.7 [Complex statements], page 78)
9. methods for outputting simple statements (e.g., communication statements; see Section 4.4 [Communication statements], page 63)
10. methods for outputting nodes with non-textual names (e.g., '`...`' and various operators)

The `NCPTL_CodeGen` class defined in '`codegen_c_generic.py`' generates code as follows. The `generate` method, which is invoked by '`ncptl.py`', calls upon SPARK to process the abstract-syntax tree (AST) in postorder fashion. `NCPTL_CodeGen` maintains a stack (`codestack`) on which code fragments are pushed and popped but that ends up containing a complete line of C code in each element. For example, in the coNCePTuaL program '`TASK 0 OUTPUTS 1+2*3`', the `n_outputs` method will pop '`[('expr', '(1)+((2)*(3))')]`' (a list

containing the single expression '`1+2*3`') and '`('task_expr', '0')`' (a tuple designating a task by the expression '`0`') and push multiple lines of code that prepare task 0 to evaluate and output the given expression.

The utility functions are the most useful for backend developers to understand, as they are frequently called from hook methods (see Section 6.2.1 [Hook methods], page 95). The following should be of particular importance:

`push`
`pushmany` Push a single value (typically a string of C code) or each value in a list of values onto a stack.

`error_fatal`
`error_internal`
Output a generic error message or an "internal error" error message and abort the program.

`code_declare_var`
Push (using the `push` method) a line of C code that declares a variable with an optionally specified type, name, initial value, and comment. Return the variable name actually used.

See the definitions in '`codegen_c_generic.py`' of each of the above to determine required and optional parameters. The following, adapted from '`codegen_c_udgram.py`' demonstrates some of the preceding methods:

```
def n_for_count_SYNC_ALL(self, localvars):
    "Synchronize all of the tasks in the job."
    synccode = []
    self.push("{", synccode)
    loopvar = self.code_declare_var(suffix="task",
      comment="Loop variable that iterates over all (physical) ranks",
      stack=synccode)
    self.pushmany([
      "thisev_sync->s.sync.peerqueue = ncptl_queue_init (sizeof(int));",
      "for (%s=0; %s<var_num_tasks; %s++)" %
      (loopvar, loopvar, loopvar),
      "*(int *)ncptl_queue_allocate(thisev_sync->s.sync.peerqueue) = %s;" %
      loopvar,
      "thisev_sync->s.sync.syncrank = physrank;",
      "}"],
                  stack=synccode)
    return synccode
```

That definition of the `n_for_count_SYNC_ALL` hook method defines a new stack (`synccode`) and pushes a '`{`' onto it. It then declares a loop variable, letting `code_declare_var` select a name but dictating that it end in '`_task`'. The hook method then pushes some additional C code onto the `synccode` stack and finally returns the stack (which is really just a list of lines of C code).

Some useful variables defined by `NCPTL_CodeGen` include the following:

`base_global_parameters`
: a list of 6-ary tuples defining extra command-line parameters to parse (format: {*type*, *variable*, *long_name*, *short_name*, *description*, *default_value*})

`events_used`
: a dictionary containing the names of events actually used by the program being compiled

Some methods in '`codegen_c_generic.py`' that are worth understanding but are unlikely to be used directly in a derived backend include the following:

`pop`
: Pop a value from a stack.

`push_marker`
: Push a specially designated "marker" value onto a stack.

`combine_to_marker`
: Pop all items off a stack up to the first marker value found; discard the marker; then, push the popped items as a single list of items. This is used, for example, by a complex statement (see Section 4.7 [Complex statements], page 78) that applies to a list of statements, which can be popped as a unit using `combine_to_marker`.

`invoke_hook`
: Call a hook method, specifying code to be pushed before/after the hook-produced code and alternative text (or Python code) to be pushed (or executed) in the case that a hook method is not provided.

6.3 Run-time library functions

To simplify the backend developer's task and to provide consistent functionality across backends, coNCePTuaL provides a run-time library that encapsulates many of the common operations needed for network-correctness and performance-testing programs. This section describes all of the functions that the library exports (plus a few important types and variables). The library is written in C, so all of the type/variable/function prototypes are expressed with C syntax. The library includes, among others, functions that manage heap-allocated memory, accurately read the time, write results to log files, control queues of arbitrary data, and implement various arithmetic operations. All of these functions should be considered "slow" and should therefore generally not be invoked while execution is being timed.[1]

6.3.1 Variables and data types

The following variables and data types are used by various run-time library functions and directly by backends.

[1] Some notable exceptions are the functions described in Section 6.3.5 [Message-buffer manipulation functions], page 106, which implement coNCePTuaL's `WITH DATA TOUCHING` and `WITH VERIFICATION` constructs.

ncptl_int Data type

The internal data type of the coNCePTuaL run-time library is `ncptl_int`. This is normally a 64-bit signed integer type selected automatically by '`configure`' (see Section 2.1 [configure], page 4) but can be overridden with the *`--with-datatype`* option to '`configure`'. '`ncptl.h`' defines a string macro called `NICS` that can be used to output an `ncptl_int` regardless of how the `ncptl_int` type is declared:

```
ncptl_fatal ("My variable contains a negative value (%" NICS ")",
             my_ncptl_int_var);
```

`ncptl_int` constants declared by backends derived from '`codegen_c_generic.py`' are given an explicit suffix which defaults to '`LL`' but can be overridden at configuration time using the *`--with-const-suffix`* option.

NCPTL_CMDLINE Data type

The `NCPTL_CMDLINE` structure describes an acceptable command-line option. It contains a type, which is either `NCPTL_TYPE_INT` for an `ncptl_int` or `NCPTL_TYPE_STRING` for a `char *`, a pointer to a variable that will receive the value specified on the command line, the long name of the argument (without the '`--`'), the one-letter short name of the argument (without the '`-`'), a textual description of what the argument represents, and a default value to use if the option is not specified on the command line.

NCPTL_QUEUE Data type

An `NCPTL_QUEUE` is an opaque data type that represents a dynamically growing queue that can be flattened to an array for more convenient access. `NCPTL_QUEUE`s have proved to be quite useful when implementing coNCePTuaL backends.

NCPTL_LOG_FILE_STATE Data type

Every coNCePTuaL log file is backed by a unique `NCPTL_LOG_FILE_STATE` opaque data type. An `NCPTL_LOG_FILE_STATE` data type represents all of the state needed to maintain that file, such as file descriptors, header comments, and data which has not yet been aggregated.

`int` **ncptl_pagesize** Variable

This variable is initialized by `ncptl_init()` to the number of bytes in an operating-system memory page. `ncptl_pagesize` can be used by backends to implement coNCePTuaL's `PAGE SIZED` and `PAGE ALIGNED` keywords (see Section 4.4.1 [Message specifications], page 63).

6.3.2 Miscellaneous functions

The following functions do not fit neatly into any of the remaining categories.

`void` **ncptl_fatal** (`char *`*format*`, ...`) Function

Output an error message and abort the program. `ncptl_fatal()` takes the same types of arguments as C's `printf()` routine.

`void` **ncptl_touch_memory** (`void *`*buffer*`, ncptl_int` *numbytes*`, ncptl_int` *bytestride*`, ncptl_int` *accesses*`, ncptl_int` *wordsize*) Function

Walk a memory region of a given size with a given stride (or '`-1`' for random) for a given number of accesses and using a given access size. For example, '`ncptl_touch_memory (mybuffer, 1048576, 4096, 10000, 64)`' will read (and do nothing with) the first 64 bytes of a 1 MB memory region, then the 64 bytes starting at offset 4096, then the 64 bytes starting at 4096 × 2, then at 4096 × 3, then at 4096 × 4, and so forth up to 4096 × 10000, wrapping around the 1 MB region as necessary. `ncptl_touch_memory()` is intended to to be used to implement the `TOUCHES` statement (see Section 4.6.4 [Touching memory], page 75).

`ncptl_int` **ncptl_assign_processor** (`ncptl_int` *virtID*`, ncptl_int` *physID*`, ncptl_int *`*virt2phys*`, ncptl_int` *numtasks*`, ncptl_int` *physrank*) Function

Assign a (physical) processor ID, *physID*, to a (virtual) task ID, *virtID* given a virtual-to-physical mapping table, *virt2phys*, and its length, *numtasks*. Return a new task ID for our processor given its processor number, *physrank*. `ncptl_assign_processor()` is intended to implement the `IS ASSIGNED TO` construct (see Section 4.6.5 [Reordering task IDs], page 76).

6.3.3 Initialization functions

The following functions are intended to be called fairly early in the generated code.

`void` **ncptl_init** (`int` *version*`, char *`*program_name*) Function

Initialize the coNCePTuaL run-time library. *version* is used to verify that '`runtimelib.c`' corresponds to the version of '`ncptl.h`' used by the generated code. The caller must pass in `NCPTL_RUN_TIME_VERSION` for *version*. *program_name* is the name of the executable program and is used for outputting error messages. The caller should pass in `argv[0]` for *program_name*. `ncptl_init()` must be the first library function called by the generated code (with a few exceptions, as indicated below).

`void` **ncptl_permit_signal** (`int` *signalnum*) Function

Indicate that the backend relies on signal *signalnum* for correct operation. Because signal handling has performance implications, the coNCePTuaL run-time library normally terminates the program upon receiving a signal. Hence, the user can be assured that if a program runs to completion then no signals have affected its performance. See Section 3.3 [Running coNCePTuaL programs], page 21, for a description of the *`--no-trap`* command-line option, which enables a user to permit additional signals to be delivered to the program (e.g., when linking with a particular implementation of a communication library that relies on signals). `ncptl_permit_signal()` must be invoked before `ncptl_parse_command_line()` to have any effect.

`void` **ncptl_parse_command_line** (`int` *argc*`, char *`*argv*`[],` `NCPTL_CMDLINE *`*arglist*`, int` *numargs*) Function

Parse the command line. *argc* and *argv* should be the argument count and argument vector passed to the generated code by the operating system. *arglist* is a list of

descriptions of acceptable command-line arguments and *numargs* is the length of that list.

Because `ncptl_init()` takes many seconds to run, it is common for generated code to scan the command line for `--help` or `-?` and, if found, skip `ncptl_init()` and immediately invoke `ncptl_parse_command_line()`. Doing so gives the user immediate feedback when requesting program usage information. Skipping `ncptl_init()` is safe in this context because `ncptl_parse_command_line()` terminates the program after displaying usage information; it does not require any information discovered by `ncptl_init()`.

Most generated programs have a `--seed`/`-S` option that enables the user to specify explicitly a seed for the random-number generator with `--help`/`-?` showing the default seed. `ncptl_seed_random_task()` must therefore be called before `ncptl_parse_command_line()` which, as stated in the previous paragraph, can be invoked without a prior invocation of `ncptl_init()`. Consequently, it can be considered safe also to invoke `ncptl_seed_random_task()` before `ncptl_init()`.

A generated program's initialization routine will generally exhibit a structure based on the following pseudocode:

```
if "--help" or "-?" in command-line options then
  only_help := TRUE
else
  only_help := FALSE
  ncptl_init(...)
end if
random_seed := ncptl_seed_random_task(0)
ncptl_parse_command_line(...)
if only_help = TRUE then
  ncptl_error("Internal error; should have exited")
end if
ncptl_seed_random_task(random_seed)
```

6.3.4 Memory-allocation functions

The coNCePTuaL run-time library provides its own wrappers for `malloc()`, `free()`, `realloc()`, and `strdup()` as well as a specialized `malloc()` designed specifically for allocating message buffers. The wrappers' "value added" is that they support the explicit data alignments needed by `ALIGNED` messages (see Section 4.4.1 [Message specifications], page 63) and that they automatically call `ncptl_fatal()` on failure, so the return value does not need to be checked for `NULL`.

`void *` **ncptl_malloc** (`ncptl_int` *numbytes*, `ncptl_int` *alignment*) Function

Allocate *numbytes* bytes of memory aligned to an *alignment*-byte boundary. If *alignment* is '`0`', `ncptl_malloc()` will use whatever alignment is "natural" for the underlying architecture. `ncptl_malloc()` will automatically call `ncptl_fatal()` if memory allocation fails. Therefore, unlike `malloc()`, there is no need to check the return value for `NULL`.

`void` **ncptl_free** (`void` **pointer*) Function

Free memory previously allocated by `ncptl_malloc()`. It is an error to pass `ncptl_free()` memory not allocated by `ncptl_malloc()`.

`void *` **ncptl_realloc** (`void` **pointer*, `ncptl_int` *numbytes*, `ncptl_int` *alignment*) Function

Given a pointer returned by `ncptl_malloc()`, change its size to *numbytes* and byte-alignment to *alignment* without altering the contents (except for truncation in the case of a smaller target size). If *alignment* is '`0`', `ncptl_realloc()` will use whatever alignment is "natural" for the underlying architecture. `ncptl_realloc()` will automatically call `ncptl_fatal()` if memory allocation fails. Therefore, unlike `realloc()`, there is no need to check the return value for `NULL`.

`char *` **ncptl_strdup** (`const char` **string*) Function

`ncptl_strdup()` copies a string as does the standard C `strdup()` function. However, `ncptl_strdup()` uses `ncptl_malloc()` instead of `malloc()` to allocate memory for the copy, which must therefore be deallocated using `ncptl_free()`.

`void *` **ncptl_malloc_message** (`ncptl_int` *numbytes*, `ncptl_int` *alignment*, `ncptl_int` *outstanding*) Function

Allocate *numbytes* bytes of memory from the heap aligned on an *alignment*-byte boundary. All calls with the same value of *outstanding* will share a buffer. `ncptl_malloc_message()` is intended to be used in two passes. The first time the function is called on a set of messages it merely determines how much memory to allocate. The second time, it returns valid memory buffers. Note that the returned pointer can be neither `free()`d nor `ncptl_free()`d.

6.3.5 Message-buffer manipulation functions

The coNCePTuaL language facilitates verifying message contents and touching every word in a message (see Section 4.4.1 [Message specifications], page 63). The following functions implement those features.

`void` **ncptl_fill_buffer** (`void` **buffer*, `ncptl_int` *numbytes*, `int` *validity*) Function

Fill a region of memory with known values. If *validity* is '`+1`', `ncptl_fill_buffer()` will fill the first *numbytes* bytes of *buffer* with a verifiable sequence of integers (see Section 4.4.1 [Message specifications], page 63). If *validity* is '`-1`', `ncptl_fill_buffer()` will pollute the first *numbytes* bytes of *buffer*. Receive buffers should be polluted before reception to avoid false negatives caused, for example, by an inadvertently dropped message destined for a previously validated buffer.

`ncptl_int` **ncptl_verify** (`void` **buffer*, `ncptl_int` *numbytes*) Function

Verify the contents of memory filled by `ncptl_fill_buffer()`. The function returns the number of erroneous bits. `ncptl_verify()` is used to implement coNCePTuaL's `WITH VERIFICATION` construct (see Section 4.4.1 [Message specifications], page 63).

`void` **ncptl_touch_data** (`void *`*buffer*`, ncptl_int` *numbytes*) Function

Touch every byte in a given buffer. `ncptl_touch_data()` is used to implement the `WITH DATA TOUCHING` construct described in Section 4.4.1 [Message specifications], page 63.

6.3.6 Time-related functions

An essential component of any benchmarking system is an accurate timer. coNCePTuaL's `ncptl_time()` function selects from a variety of timers at configuration time, first favoring lower-overhead cycle-accurate timers, then higher-overhead cycle-accurate, and finally non-cycle-accurate timers. `ncptl_init()` measures the actual timer overhead and resolution and `ncptl_log_write_header()` writes this information to the log file. Furthermore, the '`validatetimer`' program (see Section 2.2 [make], page 5) can be used to verify that the timer used by `ncptl_init()` truly does correspond to wall-clock time.

The coNCePTuaL language provides a few time-related functions. These are also supported by the functions described below.

`uint64_t` **ncptl_time** (`void`) Function

Return the time in microseconds. The timer ticks even when the program is not currently scheduled. No assumptions can be made about the relation of the value returned to the time of day; `ncptl_time()` is intended to be used strictly for computing elapsed time. The timer's resolution and accuracy are logged to the log file by `ncptl_log_write_header()` (more precisely, by the internal `log_write_header_timer()` function, which is called by `ncptl_log_write_header()`). Note that `ncptl_time()` always returns a 64-bit unsigned value, regardless of how `ncptl_int` is declared.

`void` **ncptl_set_flag_after_usecs** (`volatile int *`*flag*`, uint64_t` *delay*) Function

`ncptl_set_flag_after_usecs()` uses the operating system's interval timer to asynchronously set a variable to '`1`' after a given number of microseconds. This function is intended to be used to support the '`FOR` *time*' construct (see Section 4.7.2 [Iterating], page 79). Note that *delay* is a 64-bit unsigned value, regardless of how `ncptl_int` is declared.

`void` **ncptl_udelay** (`uint64_t` *delay*`, int` *spin0block1*) Function

If *spin0block1* is '`0`', `ncptl_udelay()` spins for *delay* microseconds (i.e., using the CPU). If *spin0block1* is '`1`', `ncptl_udelay()` sleeps for *delay* microseconds (i.e., relinquishing the CPU). Note that *delay* is a 64-bit unsigned value, regardless of how `ncptl_int` is declared. `ncptl_udelay()` is intended to be used to support the coNCePTuaL language's `SLEEPS` and `COMPUTES` statements (see Section 4.6.3 [Delaying execution], page 74).

6.3.7 Log-file functions

Benchmarking has limited value without a proper record of the performance results. The coNCePTuaL run-time library provides functions for writing data to log files. It

takes care of much of the work needed to calculate statistics on data columns and to log a thorough experimental setup to every log file.

The library treats a log file as a collection of tables of data. Each table contains a number of rows, one per dynamic invocation of the `LOGS` statement (see Section 4.5.3 [Writing to a log file], page 71). Each row contains a number of columns, one per aggregate expression (see Section 4.2.3 [Aggregate expressions], page 59) expressed statically in a coNCePTuaL program.[2] Log-file functions should be called only if the coNCePTuaL source code accesses a log file (see Section 4.5.3 [Writing to a log file], page 71).

`void` **ncptl_log_add_comment** (`const char *`*key*`, const char *`*value*) Function

`ncptl_log_add_comment()` makes it possible for a backend to add backend-specific ⟨*key*:*value*⟩ pairs to the set of header comments that get written to a log file (see Section 3.4.1 [Log-file format], page 23). `ncptl_log_add_comment()` can be called repeatedly; all calls should precede `ncptl_log_open()` to have any effect. Note that `ncptl_log_add_comment()` makes a copy of *key* and *value*, so these need not be heap-allocated.

`NCPTL_LOG_FILE_STATE *` **ncptl_log_open** (`char *`*template*`, ncptl_int` *processor*) Function

Given a filename template containing a '`%d`' placeholder and a processor number (i.e., the process's physical rank in the computation), `ncptl_log_open()` creates and opens a log file named by the template with '`%d`' replaced by *processor*. For example, if *template* is '`/home/me/myprog-%d.log`' and *processor* is '`3`', the resulting filename will be '`/home/me/myprog-3.log`'. `ncptl_log_open()` must be called before any of the other `ncptl_log_`*something*`()` functions—except for `ncptl_log_add_comment()`, which should be called before `ncptl_log_open()`. `ncptl_log_open()` returns a pointer to an opaque `NCPTL_LOG_FILE_STATE` value; the backend will need to pass this pointer to nearly all of the other log-file functions described in this section.

`void` **ncptl_log_write_header** (`NCPTL_LOG_FILE_STATE *`*logstate*`, char *`*progname*`, char *`*backend_name*`, char *`*backend_desc*`, int` *processor*`, int` *numtasks*`, NCPTL_CMDLINE *`*arglist*`, int` *numargs*`, char **`*sourcecode*) Function

`ncptl_log_write_header()` standardizes the header with which all log files begin. *progname* is the name of the program executable (`argv[0]` in C). *backend_name* is the name of the backend in '*language_library*' format (e.g., '`java_rmi`'). *backend_desc* is a brief description of the backend (e.g., '`Java + RMI`'). *processor* is the caller's physical rank in the program (the same value passed to `ncptl_log_open()`). *numtasks* is the total number of tasks in the program. *arglist* is the list of arguments passed to `ncptl_parse_command_line()` and *numargs* is the number of entries in that list. *sourcecode* is the complete coNCePTuaL source code stored as a `NULL`-terminated list of `NULL`-terminated strings.

[2] Writing `A HISTOGRAM OF THE` ⟨*expr*⟩ produces two columns, one for values and one for tallies.

`void` **ncptl_log_write** (`NCPTL_LOG_FILE_STATE *`*logstate*`, int` *logcolumn*`, char *`*description*`, LOG_AGGREGATE` *aggregate*`, double` *value*) Function

Push value *value* onto column *logcolumn* of the current table. Gaps between columns are automatically elided. *description* is used as the column header for column *logcolumn*. Acceptable values for *aggregate* are defined in Section B.3 [Representing aggregate functions], page 133.

`void` **ncptl_log_compute_aggregates** (`NCPTL_LOG_FILE_STATE *`*logstate*) Function

`ncptl_log_compute_aggregates()` implements the `COMPUTES AGGREGATES` construct described in Section 4.5.3 [Writing to a log file], page 71. When `ncptl_log_compute_aggregates()` is invoked, the coNCePTuaL run-time library uses the aggregate function specified by `ncptl_log_write()` to aggregate all of the data that accumulated in each column since the last invocation of `ncptl_log_compute_aggregates()`. Note that `ncptl_log_compute_aggregates()` is called implicitly by `ncptl_log_commit_data()`.

`void` **ncptl_log_commit_data** (`NCPTL_LOG_FILE_STATE *`*logstate*) Function

The coNCePTuaL run-time library keeps the current data table in memory and doesn't write anything to the log file until `ncptl_log_commit_data()` is called, at which point the run-time library writes all accumulated data to the log file and begins a new data table. Note that `ncptl_log_commit_data()` is called implicitly by `ncptl_log_close()`. Furthermore, a backend should call `ncptl_log_commit_data()` when beginning execution of a new statement in a coNCePTuaL program. For instance, the '`codegen_c_generic.py`' backend invokes `ncptl_log_commit_data()` from `code_def_main_newstmt`.

`void` **ncptl_log_write_footer** (`NCPTL_LOG_FILE_STATE *`*logstate*) Function

`ncptl_log_write_footer()` write a stock footer to the log file.

`void` **ncptl_log_close** (`NCPTL_LOG_FILE_STATE *`*logstate*) Function

Close the log file. No `ncptl_log_`*something*`()` function should be called after `ncptl_log_close()` is invoked.

6.3.8 Random-task functions

Randomness appears in various forms in the coNCePTuaL language, such as when assigning a task to `A RANDOM PROCESSOR` (see Section 4.6.5 [Reordering task IDs], page 76) or when let-binding `A RANDOM TASK` or `A RANDOM TASK OTHER THAN` a given task ID to a variable (see Section 4.7.3 [Binding variables], page 83). The following functions are used to select tasks at random. coNCePTuaL currently uses the Mersenne Twister as its random-number generator. Hence, given the same seed, a coNCePTuaL program will see the same random-number sequence on every platform.

`int` **ncptl_seed_random_task** (`int` *seed*) Function

Initialize the random-number generator needed by `ncptl_random_task()`. If *seed* is zero, `ncptl_seed_random_task()` selects an arbitrary seed value. `ncptl_seed_random_task()` returns the seed that was used.

`ncptl_int` **ncptl_random_task** (`ncptl_int` *lower_bound*, `ncptl_int` *upper_bound*, `ncptl_int` *excluded*) Function

Return a randomly selected task number from *lower_bound* to *upper_bound* (both inclusive). If *excluded* is nonnegative then that task number will never be selected, even if it's within range.

6.3.9 Queue functions

Because queues are a widely applicable construct, the run-time library provides support for queues of arbitrary datatypes. In the current implementation, these can more precisely be termed "dynamically growing lists" than "queues". However, they may be extended in a future version of the library to support more queue-like functionality.

`NCPTL_QUEUE *` **ncptl_queue_init** (`ncptl_int` *eltbytes*) Function

`ncptl_queue_init()` creates and initializes a dynamically growing queue in which each element occupies *eltbytes* bytes of memory.

`void *` **ncptl_queue_allocate** (`NCPTL_QUEUE *`*queue*) Function

Allocate a new data element at the end of queue *queue*. The queue passed to `ncptl_queue_allocate()` must be one returned by `ncptl_queue_init()`. `ncptl_queue_allocate()` returns a pointer to the data element allocated.

`void *` **ncptl_queue_push** (`NCPTL_QUEUE *`*queue*, `void *`*element*) Function

Push (via a memory copy) the element pointed to by *element* onto the end of queue *queue* and return a pointer to the copy in the queue. The queue passed to `ncptl_queue_allocate()` must be one returned by `ncptl_queue_init()`. (`ncptl_queue_push()` is actually implemented in terms of `ncptl_queue_allocate()`.)

`void *` **ncptl_queue_pop** (`NCPTL_QUEUE *`*queue*) Function

Pop a pointer to the element at the head of queue *queue*. If *queue* is empty, return `NULL`. The pointer returned by `ncptl_queue_pop()` is guaranteed to be valid until the next invocation of `ncptl_queue_free()`.

`void *` **ncptl_queue_contents** (`NCPTL_QUEUE *`*queue*, `int` *copyelts*) Function

Return queue *queue* as an array of elements. If `ncptl_queue_contents()` is passed '`1`' for *copyelts*, a new array is allocated using `ncptl_malloc()`; the queue's internal array is copied to the newly allocated array; and, this new array is returned to the caller. It is the caller's responsibility to pass the result to `ncptl_free()` when the array is no longer needed. If `ncptl_queue_contents()` is passed '`0`' for *copyelts*, a pointer to the queue's internal array is returned without first copying it. This pointer should not be passed to `ncptl_free()` as it is still needed by *queue*.

`ncptl_int` **ncptl_queue_length** (`NCPTL_QUEUE *`*queue*) Function

Return the number of elements in queue *queue*.

`void` **ncptl_queue_empty** (`NCPTL_QUEUE *`*queue*) Function

Empty a queue, freeing the memory it had previously used. *queue* should not be used after a call to `ncptl_queue_empty()`. Queue contents returned by `ncptl_queue_contents()` with *copyelts* set to '`0`' are also invalidated.

6.3.10 Language-visible functions

The coNCePTuaL language contains a number of built-in functions that perform various operations on floating-point numbers (used when writing to a log file or the standard output device) and integers (used at all other times) and that determine the IDs of neighboring tasks on a variety of topologies. Each function occurs in two forms: `ncptl_func_`*function*, which maps `ncptl_int`s to `ncptl_int`s, and `ncptl_dfunc_`*function*, which maps `double`s to `double`s. See Section 4.2.2 [Built-in functions], page 49, for additional details about each function's semantics.

Although some of the functions described in this section are fairly simple, including them in the run-time library ensures that each function returns the same value across different backends and across different platforms.

Integer functions

`ncptl_int` **ncptl_func_sqrt** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_sqrt** (`double` *num*) Function

`ncptl_func_sqrt()` returns the unique integer x such that $x^2 \leq num \wedge (x+1)^2 > num$. `ncptl_dfunc_sqrt()` returns $\sqrt{num}$ in double-precision arithmetic.

`ncptl_int` **ncptl_func_cbrt** (`ncptl_int` *num*) Function

`ncptl_func_cbrt()` returns the unique integer x such that $x^3 \leq num \wedge (x+1)^3 > num$. `ncptl_dfunc_cbrt()` returns $\sqrt[3]{num}$ in double-precision arithmetic.

`ncptl_int` **ncptl_func_bits** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_bits** (`double` *num*) Function

Return the minimum number of bits needed to represent a given integer. (*num* is rounded up to the nearest integer in the case of `ncptl_dfunc_bits()`.)

`ncptl_int` **ncptl_func_log10** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_log10** (`double` *num*) Function

Return $\log_{10}(num)$ In the case of `ncptl_func_log10()`, this value is rounded down to the nearest integer.

`ncptl_int` **ncptl_func_factor10** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_factor10** (`double` *num*) Function

Return *num* rounded down to the nearest single-digit factor of a power of 10.

`ncptl_int` **ncptl_func_abs** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_abs** (`double` *num*) Function

Return $|num|$. In the case of `ncptl_func_log10()`, this value is rounded down to the nearest integer.

`ncptl_int` **ncptl_func_power** (`ncptl_int` *base*, `ncptl_int` *exponent*) Function
`double` **ncptl_dfunc_power** (`double` *base*, `double` *exponent*) Function

Return *base* raised to the power of *exponent*.

`ncptl_int` **ncptl_func_modulo** (`ncptl_int` *numerator*, `ncptl_int` *denominator*) Function
`double` **ncptl_dfunc_modulo** (`double` *numerator*, `double` *denominator*) Function

Return the remainder of dividing *numerator* by *denominator*. The result is guaranteed to be a nonnegative integer. `ncptl_dfunc_modulo()` rounds each of *numerator* and *denominator* down to the nearest integer before dividing and taking the remainder.

Floating-point functions

`ncptl_int` **ncptl_func_floor** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_floor** (`double` *num*) Function

Return $\lfloor num \rfloor$. (This is the identity function in the case of `ncptl_func_floor()`.)

`ncptl_int` **ncptl_func_ceiling** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_ceiling** (`double` *num*) Function

Return $\lceil num \rceil$. (This is the identity function in the case of `ncptl_func_ceiling()`.)

`ncptl_int` **ncptl_func_round** (`ncptl_int` *num*) Function
`double` **ncptl_dfunc_round** (`double` *num*) Function

Return *num* rounded to the nearest integer. (This is the identity function in the case of `ncptl_func_round()`.)

Topology functions

In the following functions, the '`dfunc`' versions merely cast their arguments to `ncptl_ints` and call the corresponding '`func`' versions.

`ncptl_int` **ncptl_func_tree_parent** (`ncptl_int` *task*, `ncptl_int` *arity*) Function
`double` **ncptl_dfunc_tree_parent** (`double` *task*, `double` *arity*) Function

Return task *task*'s parent in an *arity*-ary tree.

`ncptl_int` **ncptl_func_tree_child** (`ncptl_int` *task*, `ncptl_int` *child*, `ncptl_int` *arity*) Function
`double` **ncptl_dfunc_tree_child** (`double` *task*, `double` *child*, `double` *arity*) Function

Return child *child* of task *task* in an *arity*-ary tree.

`ncptl_int` **ncptl_func_grid_coord** (`ncptl_int vartask`, `ncptl_int` *coord*, `ncptl_int` *width*, `ncptl_int` *height*, `ncptl_int` *depth*) Function
`double` **ncptl_dfunc_grid_coord** (`double vartask`, `double` *coord*, `double` *width*, `double` *height*, `double` *depth*) Function

Return task *task*'s x coordinate (*coord* = 0), y coordinate (*coord* = 1), or z coordinate (*coord* = 2) on a *width* × *height* × *depth* mesh (or torus).

`ncptl_int` **ncptl_func_grid_neighbor** (`ncptl_int` *task*, `ncptl_int` *torus*, `ncptl_int` *width*, `ncptl_int` *height*, `ncptl_int` *depth*, `ncptl_int` *xdelta*, `ncptl_int` *ydelta*, `ncptl_int` *zdelta*) Function

`double` **ncptl_dfunc_grid_neighbor** (`double` *task*, `double` *torus*, `double` *width*, `double` *height*, `double` *depth*, `double` *xdelta*, `double` *ydelta*, `double` *zdelta*) Function

Return one of task *task*'s neighbors—not necessarily an immediate neighbor—on a 3-D mesh or torus. For the following explanation, assume that task *task* lies at coordinates (x, y, z) on a $width \times height \times depth$ mesh or torus. In the mesh case ($torus = 0$), the value returned is the task ID corresponding to coordinates $(x + xdelta, y + ydelta, z + zdelta)$. In the torus case ($torus = 1$), the value returned is the task ID corresponding to coordinates $((x + xdelta) \bmod width, (y + ydelta) \bmod height, (z + zdelta) \bmod depth)$.

Note that there are no 1-D or 2-D grid functions. Instead, the appropriate 3-D function should be used with *depth* and—in the 1-D case—*height* set to '`1`'.

`ncptl_int` **ncptl_func_knomial_parent** (`ncptl_int` *task*, `ncptl_int` *arity*, `ncptl_int` *numtasks*) Function

`double` **ncptl_dfunc_knomial_parent** (`double` *task*, `double` *arity*, `double` *numtasks*) Function

Return task *task*'s parent in an *arity*-nomial tree of *numtasks* tasks.

`ncptl_int` **ncptl_func_knomial_child** (`ncptl_int` *task*, `ncptl_int` *child*, `ncptl_int` *arity*, `ncptl_int` *numtasks*, `ncptl_int` *count_only*) Function

`double` **ncptl_dfunc_knomial_child** (`double` *task*, `double` *child*, `double` *arity*, `double` *numtasks*, `double` *count_only*) Function

If *count_only* is '`0`', return task *task*'s *child*th child in an *arity*-nomial tree of *numtasks* tasks. If *count_only* is '`1`', return the number of children task *task* has in an *arity*-nomial tree of *numtasks* tasks.

Random-number functions

`ncptl_int` **ncptl_func_random_uniform** (`ncptl_int` *lower_bound*, `ncptl_int` *upper_bound*) Function

`double` **ncptl_dfunc_random_uniform** (`double` *lower_bound*, `double` *upper_bound*) Function

Return a number in the interval [*lower_bound*, *upper_bound*) selected at random with a uniform distribution.

`ncptl_int` **ncptl_func_random_gaussian** (`ncptl_int` *mean*, `ncptl_int` *stddev*) Function

`double` **ncptl_dfunc_random_gaussian** (`double` *mean*, `double` *stddev*) Function

Return a number selected at random from a Gaussian distribution with mean *mean* and standard deviation *stddev*.

`ncptl_int` **ncptl_func_random_poisson** (`ncptl_int` *mean*) Function
`double` **ncptl_dfunc_random_poisson** (`double` *mean*) Function

Return an integer selected at random from a Poisson distribution with mean *mean* and standard deviation $\sqrt{mean}$.

6.3.11 Finalization functions

The following function should be called towards the end of the generated code's execution.

`void` **ncptl_finalize** (`void`) Function

Shut down the coNCePTuaL run-time library. No run-time library functions should be invoked after `ncptl_finalize()`.

Among other operations, `ncptl_finalize()` resets all signal handlers to their original values and kills the watchdog process, if any. (See Section 3.3 [Running coNCePTuaL programs], page 21.) Because the watchdog mechanism is based on trapping `SIGCHLD` signal, it is important that `ncptl_finalize()` be called *before* the termination of any `fork()`'ed processes. Otherwise, an undesired watchdog interrupt will kill the parent process.

7 Tips and Tricks

The following sections present some ways to make better use of coNCePTuaL in terms of producing simpler, more efficient programs.

7.1 Using out-of-bound task IDs to simplify code

See Section 4.3 [Task descriptions], page 61, mentions a language feature that can substantially simplify coNCePTuaL programs: Operations involving out-of-bound task IDs are silently ignored. The beauty of this feature is that it reduces the need for special cases at network boundaries. Consider, for example, a simple pipeline pattern in which each task in turn sends a message to the subsequent task:

```
ALL TASKS t SEND A 64 DOUBLEWORD MESSAGE TO TASK t+1.
```

Because implicit receives are posted before the corresponding sends (see Section 4.4.2 [Sending], page 67), all tasks except task 0 start by posting a blocking receive. (No task is sending to task 0.) Task 0 is therefore free to send a message to task 1. Receipt of that message unblocks task 1, who then sends a message to task 2, thereby unblocking task 3, and so forth. Without needing an explicit special case in the program, task '`num_tasks-1`' receives a message from task '`num_tasks-2`' but does not attempt to send a message to nonexistent task '`num_tasks`', thanks to the rule that communication with nonexistent tasks turns into a no-op (i.e., is elided from the program).

As a more complex variation of the same program, consider a wavefront communication pattern that progresses from the upper-left corner of a mesh to the lower-right corner. Such a pattern can be expressed in just four lines of coNCePTuaL (receive left, receive up, send right, send down) by relying on the property that communication with a nonexistent task is simply not executed:

```
TASK MESH_NEIGHBOR(src, xsize, +1, ysize,  0) RECEIVES A
  64 DOUBLEWORD MESSAGE FROM ALL TASKS src THEN
TASK MESH_NEIGHBOR(src, xsize,  0, ysize, +1) RECEIVES A
  64 DOUBLEWORD MESSAGE FROM ALL TASKS src THEN
ALL TASKS src SEND A 64 DOUBLEWORD MESSAGE TO
  UNSUSPECTING TASK MESH_NEIGHBOR(src, xsize, +1, ysize,  0) THEN
ALL TASKS src SEND A 64 DOUBLEWORD MESSAGE TO
  UNSUSPECTING TASK MESH_NEIGHBOR(src, xsize,  0, ysize, +1).
```

To understand the preceding program recall that `MESH_NEIGHBOR` returns '`-1`' for nonexistent neighbors. Because '`-1`' is outside of the range [0, `num_tasks`) communication with a nonexistent neighbor is ignored. To help the reader understand the preceding program, we present a trace of the events it posts as it runs with a 2×2 arrangement of tasks:

```
[TRACE] phys: 0 | virt: 0 | action: SEND | event: 1 / 2 | lines: 3 - 3
[TRACE] phys: 1 | virt: 1 | action: RECV | event: 1 / 2 | lines: 1 - 1
[TRACE] phys: 2 | virt: 2 | action: RECV | event: 1 / 2 | lines: 2 - 2
[TRACE] phys: 3 | virt: 3 | action: RECV | event: 1 / 2 | lines: 1 - 1

[TRACE] phys: 0 | virt: 0 | action: SEND | event: 2 / 2 | lines: 4 - 4
[TRACE] phys: 1 | virt: 1 | action: SEND | event: 2 / 2 | lines: 4 - 4
```

```
[TRACE] phys: 2 | virt: 2 | action: SEND | event: 2 / 2 | lines: 3 - 3

[TRACE] phys: 3 | virt: 3 | action: RECV | event: 2 / 2 | lines: 2 - 2
```

The `c_trace` backend (see Section 3.2.4 [The c_trace backend], page 14) was used to produce that trace. To increase clarity, we manually added blank lines to group concurrent events (i.e., there is no significance to the order of the `TRACE` lines within each group). The important thing to notice is that there are exactly four receives and exactly four sends:

- Although all tasks are instructed to receive a message from the left, only tasks 1 and 3 actually do so;
- although all tasks are instructed to receive a message from above, only tasks 2 and 3 actually do so;
- although all tasks are instructed to send a message to the right, only tasks 0 and 2 actually do so; and,
- although all tasks are instructed to send a message downwards, only tasks 0 and 1 actually do so.

Because communication with nonexistent tasks is elided at program initialization time there is no run-time cost for such operations—as evidenced by the `c_trace` output presented above. Furthermore, there is no reliance on the backend to drop messages from nonexistent senders or to nonexistent receivers; it is perfectly safe to utilize no-op'ed communcation in any coNCePTuaL program and when using any backend.

7.2 Proper use of conditionals

coNCePTuaL supports two forms of conditional execution: conditional expressions (see Section 4.2.1 [Arithmetic expressions], page 48) and conditional statements (see Section 4.7.4 [Conditional execution], page 84). From the perspective of code readability and "thinking in coNCePTuaL" it is generally preferable to use restricted identifiers (see Section 4.3.1 [Restricted identifiers], page 61) to select groups of tasks rather than a loop with a conditional as would be typical in other programming languages. For example, consider the following code in which certain even-numbered tasks each send a message to the right:

```
FOR EACH evtask IN {0, ..., num_tasks-1}
  IF evtask IS EVEN /\ evtask MOD 3 <> 2 THEN
    TASK evtask SENDS A 64 BYTE MESSAGE TO TASK evtask+1
```

While the preceding control flow is representative of that in other programming languages, coNCePTuaL can express the same communication pattern without needing either a loop or an explicit conditional statement:

```
TASK evtask SUCH THAT evtask IS EVEN /\ evtask MOD 3 <> 2 SENDS A 64
BYTE MESSAGE TO TASK evtask+1
```

One situation in which conditional statements do not have a convenient analogue is when a program selects among multiple disparate subprograms based on a command-line parameter:

```
func IS "Operation to perform (1=op1; 2=op2; 3=op3)" AND COMES FROM
"--function" OR "-f" WITH DEFAULT 1.

ASSERT THAT "the function must be 1, 2, or 3" WITH func>=1 /\ func<=3.

IF func = 1 THEN {
  Perform operation op1.
}
OTHERWISE IF func = 2 THEN {
  Perform operation op2.
}
OTHERWISE IF func = 3 THEN {
  Perform operation op3.
}
```

7.3 Memory efficiency

As described in Section 6.2.3 [Generated code], page 98, the `c_generic` backend (and therefore all derived backends) generates programs that run by executing a sequence of events in an event list. While this form of program execution makes it possible to hoist a significant amount of computation out of the timing loop, it does imply that a program's memory requirements are proportional to the number of statements that the program executes.

coNCePTuaL's memory usage can be reduced by taking advantage of repeat counts within statements that support such a construct. The language's ⟨*send_stmt*⟩ (see Section 4.4.2 [Sending], page 67), ⟨*receive_stmt*⟩ (see Section 4.4.3 [Receiving], page 68), and ⟨*touch_stmt*⟩ (see Section 4.6.4 [Touching memory], page 75) are all examples of statements that accept repeat counts. For other statements and for groups of statements that repeat, the `FOR...REPETITIONS` statement produces a single `EV_REPEAT` event followed by a single instance of the events in the loop body. This technique is valid because coNCePTuaL knows *a priori* that every iteration is identical to every other iteration. In contrast, the more general `FOR EACH` statement can induce different behavior each iteration based on the value of the loop variable so programs must conservatively instantiate the events in the loop body for every iteration. Consider the following examples:

Least efficient:

'`FOR EACH i IN {1, ..., 1000} TASK 0 TOUCHES A 1 WORD MEMORY REGION`'
(1000 `EV_TOUCH` events on task 0)

More efficient:

'`FOR 1000 REPETITIONS TASK 0 TOUCHES A 1 WORD MEMORY REGION`'
(an `EV_REPEAT` event and an `EV_TOUCH` event on task 0)

Most efficient:

'`TASK 0 TOUCHES A 1 WORD MEMORY REGION 1000 TIMES`'
(one `EV_TOUCH` event on task 0)

Least efficient:

'`FOR EACH i IN {1, ..., 1000} TASK 0 SENDS A 32 KILOBYTE MESSAGE TO TASK 1`'
(1000 `EV_SEND` events on task 0 and 1000 `EV_RECV` events on task 1)

More efficient:

'`FOR 1000 REPETITIONS TASK 0 SENDS A 32 KILOBYTE MESSAGE TO TASK 1`'
(an `EV_REPEAT` event and an `EV_SEND` event on task 0 plus an `EV_REPEAT` event and an `EV_RECV` event on task 1)

Most efficient:

'`TASK 0 SENDS 1000 32 KILOBYTE MESSAGES TO TASK 1`'
(currently the same as the above although a future release of coNCePTuaL may reduce this to a single `EV_SEND` event on task 0 and a single `EV_RECV` event on task 1)

8 Troubleshooting

In any complex system, things are bound to go wrong. The following sections present solutions to various problems that have been encountered when building coNCePTuaL and running coNCePTuaL programs.

8.1 Interpreting configure warnings

The '`configure`' script performs a large number of tests to ensure that coNCePTuaL will compile properly and function as expected. In particular, any missing or improperly functioning feature upon which the C run-time library relies causes *`./configure`* to issue a '`not building the C run-time library`' warning. Without its run-time library, coNCePTuaL's functionality is severely limited so it's worth every effort to get *`./configure`* to build that.

Like all Autoconf scripts, '`configure`' logs detailed information to a '`config.log`' file. As a general diagnostic technique one should search for puzzling output in '`config.log`' and examine the surrounding context. For instance, on one particular system, *`./configure`* output '`no`' following '`checking if we can run a trivial program linked with "-lrt -lm -lpopt "`' and then refused to build the run-time library. The following relevant lines appeared in '`config.log`':

```
configure:12845: checking if we can run a trivial program linked with "-lrt
-lm -lpopt "
configure:12862: /usr/local/bin/gcc  -o conftest -g -O2   conftest.c -lrt
-lm -lpopt  >&5
configure:12865: $? = 0
configure:12867: ./conftest
ld.so.1: ./conftest: fatal: libpopt.so.0: open failed: No such file or
directory
./configure: line 1:  5264 Killed                  ./conftest$ac_exeext
configure:12870: $? = 137
configure: program exited with status 137
configure: failed program was:
#line 12851 "configure"
#include "confdefs.h"

int
main (int argc, char *argv[])
{
  return 0;
}
configure:12879: result: no
```

Note the error message from '`ld.so.1`' about '`libpopt.so.0`' not being found. Further investigation revealed that although '`/usr/local/bin/gcc`' knew to look in '`/usr/local/lib/`' for shared libraries, that directory was not in the search path utilized by '`ld.so.1`'. Consequently, it couldn't find '`/usr/local/lib/libpopt.so.0`'. The

solution in this case was to add '`/usr/local/lib/`' to the *`LD_LIBRARY_PATH`* environment variable before running *`./configure`* .

In general, '`config.log`' should be the first place to look when trying to interpret warnings issued by *`./configure`* . Furthermore, note that certain command-line options to *`./configure`* (see Section 2.1 [configure], page 4) may help bypass problematic operations that the script stumbles over.

8.2 Compaq compilers on Alpha CPUs

Although coNCePTuaL builds fine on Alpha-based systems when using a '`gcc`' compiler, Compaq's C compilers are sometimes problematic. For instance, the '`libncptl_wrap.c`' source file fails to compile on a system with the following versions of the operating system, C compiler, and Python interpreter:

```
% uname -a
OSF1 qsc14 V5.1 2650 alpha
% cc -V
Compaq C V6.5-011 on Compaq Tru64 UNIX V5.1B (Rev. 2650)
Compiler Driver V6.5-003 (sys) cc Driver
% python -V
Python 2.3
```

On the system that was tested, '`cc`' aborts with a set of '`Missing type specifier or type qualifier`' messages. The problem appears to be that some of Compaq's standard C header files fail to `#include` various header files they depend upon. A workaround is to insert the following C preprocessor directives in '`libncptl_wrap.c`' before the line reading '`#include "Python.h"`':

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/stat.h>
```

A second problem with Compaq compilers on Alpha-based systems occurs under Linux when using Compaq's '`ccc`' compiler:

```
% uname -a
Linux wi 2.4.21-3.7qsnet #2 SMP Fri Oct 17 14:08:00 MDT 2003 alpha unknown
% ccc -V
Compaq C T6.5-002 on Linux 2.4.21-3.7qsnet alpha
Compiler Driver T6.5-001 (Linux) cc Driver
Installed as ccc-6.5.6.002-1
Red Hat Linux release 7.2 (Enigma)
Using /usr/lib/gcc-lib/alpha-redhat-linux/2.96 (4).
```

When linking files into a shared object, '`ccc`' aborts with a '`[...]/.libs: file not recognized: Is a directory`' error message. The problem appears to be that '`libtool`' is confused about the arguments it's supposed to pass to the linker; '`libtool`' uses '`--rpath`'

(two hyphens) where the Compaq linker expects '`-rpath`' (one hyphen). As a workaround, you can edit the '`libtool`' file after running *`./configure`* but before running *`make`*. Simply replace '`--rpath`' with '`-rpath`' in the following '`libtool`' line and the problem should go away:

```
hardcode_libdir_flag_spec="\${wl}--rpath \${wl}\$libdir"
```

8.3 "Cannot open shared object file" errors

By default, *`make`* will build and *`make install`* will install both a static and a dynamic version of the coNCePTuaL run time library. Most linkers give precedence to the dynamic library over the static library unless the static library is requested explicitly. As a consequence, the dynamic version of the coNCePTuaL run-time library needs to be available at program-load time in order to avoid error messages like the following:

```
a.out: error while loading shared libraries: libncptl.so.0: cannot open
shared object file: No such file or directory
```

To point the Unix dynamic loader to the coNCePTuaL run-time library merely add the directory in which '`libncptl.so`' was installed (by default, '`/usr/local/lib`') to your *`LD_LIBRARY_PATH`* environment variable. Alternatively, use the *`--disable-shared`* configuration option (see Section 2.1 [configure], page 4) to prevents coNCePTuaL from building the dynamic version of the run-time library altogether, thereby forcing the linker to use the static version:

```
make uninstall
make clean
./configure --disable-shared ...
make
make install
```

As mentioned in Section 2.1 [configure], page 4, however, '`libncptlmodule.so`' can't be built when *`--disable-shared`* is in effect.

8.4 Inhibiting the use of child processes

As of this writing, the coNCePTuaL run-time library spawns child processes for two purposes: to implement watchdog interrupts (initiated by the *`--watchdog`* option described in Section 3.3 [Running coNCePTuaL programs], page 21); and, to acquire certain pieces of system information for the log-file header (see Section 3.4.1 [Log-file format], page 23). Unfortunately, some job launchers and messaging layers get confused when processes under their control spawn other processes. For example, a child process might reinitialize shared device-driver state. The result is generally a crashed or hung program.

As a workaround, the coNCePTuaL run-time library will suppress the use of `fork()`, `system()`, `popen()`, and other process-spawning functions if the *`NCPTL_NOFORK`* environment variable is set at program-initialization time. Nonessential process-spawning functions, such as those used to produce the log-file headers, are then ignored; essential process-spawning functions, such as those that implement coNCePTuaL's watchdog interrupt, cause the

program to abort. (In other words, *--watchdog* should not be used when *NCPTL_NOFORK* is set.) If a coNCePTuaL program is inexplicably but consistently crashing or hanging, try setting *NCPTL_NOFORK* and re-running the program to see if that fixes it.

8.5 Keeping programs from dying on a signal

By default, coNCePTuaL programs terminate upon receiving *any* unexpected signal. The error message list the signal number and, if available, a human-readable signal name:

```
myprogram: Received signal 28 (Window changed)
```

The motivation behind this decision to abort on unexpected signals is that signal-handling adversely affects a program's performance. Hence, by running to completion, a program indicates that it did not receive any unexpected signals. However, some messaging layers use signals internally (most commonly `SIGUSR1` and `SIGUSR2`) to coordinate helper processes. To permit a program to deliver such signals to the messaging layer a user should use the program's *--no-trap* command-line option as described in Section 3.3 [Running coNCePTuaL programs], page 21.

8.6 "Unaligned access" warnings

On some platforms you may encounter messages like the following written to the console and/or various system log files (e.g., '`/var/log/messages`'):

```
myprog(25044): unaligned access to 0x6000000000001022,
ip=0x40000000000009e1
```

Alternatively:

```
Unaligned access pid=7890104 <myprog> va=0x140004221 pc=0x1200012b4
ra=0x1200012a4 inst=0xb449fff8
```

What's happening is that some CPUs require n-byte-wide data to be aligned on an n-byte boundary. For example, a 64-bit datatype can be accessed properly only from memory locations whose address is a multiple of 64 bits (8 bytes). On some platforms, misaligned accesses abnormally terminate the program, typically with a `SIGBUS` signal. On other platforms, misaligned accesses interrupt the operating system. The operating system fixes the access by splitting it into multiple aligned accesses plus some bit masking and shifting and then notifying the user and/or system administrator that a fixup occurred.

coNCePTuaL's '`configure`' script automatically determines what data alignments are allowed by the architecture but it has no way to determine if fixups occurred as these are transparent to programs. The result is annoying "unaligned access" messages such as those quoted above. One solution is to use the *--with-alignment* option to '`configure`' to specify explicitly the minimum data alignment that coNCePTuaL should be permitted to use. Alternatively, some operating systems provide a mechanism to cause misaligned accesses to result in a `SIGBUS` signal instead of a fixup and notification message. On Linux/IA-64 this is achieved with the command *pctrl --unaligned=signal*. On OSF1/Alpha the equivalent command is *uac p sigbus*. Be sure to rerun '`configure`' after issuing those commands to make it reexamine the set of valid data alignments.

Appendix A Reserved Words

As mentioned in Section 4.1 [Primitives], page 46, not all identifiers can be used as variables. The following sections provide a complete list of identifiers that are forbidden as variable names. These identifiers fall into two categories: keywords, which are never allowed as variable names, and predeclared variables, which are "read-only" variables; they can be utilized just like any other variables but cannot be redeclared.

A.1 Keywords

The following is a list of all currently defined keywords in the coNCePTuaL language. It is an error to try to use any of these as identifiers.

- A
- ABS
- AGGREGATES
- ALIGNED
- ALL
- AN
- AND
- ARE
- ARITHMETIC
- AS
- ASSERT
- ASSIGNED
- ASYNCHRONOUSLY
- AWAIT
- AWAITS
- BACKEND
- BE
- BIT
- BITS
- BUFFER
- BUT
- BYTE
- BYTES
- CBRT
- CEILING
- COMES
- COMPLETION
- COMPLETIONS

- COMPUTE
- COMPUTES
- COUNTERS
- DATA
- DAY
- DAYS
- DEFAULT
- DEVIATION
- DIVIDES
- DOUBLEWORD
- DOUBLEWORDS
- EACH
- EVEN
- EXECUTE
- EXECUTES
- FACTOR10
- FINAL
- FLOOR
- FOR
- FROM
- GEOMETRIC
- GIGABYTE
- GREATER
- HALFWORD
- HALFWORDS
- HARMONIC
- HISTOGRAM
- HOUR
- HOURS
- IF
- IN
- INTEGER
- INTEGERS
- INTO
- IS
- IT
- ITS
- KILOBYTE
- KNOMIAL_CHILD

- KNOMIAL_CHILDREN
- KNOMIAL_PARENT
- LANGUAGE
- LESS
- LET
- LOG
- LOG10
- LOGS
- MAX
- MAXIMUM
- MEAN
- MEDIAN
- MEGABYTE
- MEMORY
- MESH_NEIGHBOR
- MESH_COORDINATE
- MESSAGE
- MESSAGES
- MICROSECOND
- MICROSECONDS
- MILLISECOND
- MILLISECONDS
- MIN
- MINIMUM
- MINUTE
- MINUTES
- MISALIGNED
- MOD
- MULTICAST
- MULTICASTS
- NONUNIQUE
- NOT
- ODD
- OF
- OR
- OTHER
- OTHERWISE
- OUTPUT
- OUTPUTS

- PAGE
- PAGES
- PLUS
- PROCESSOR
- PROCESSORS
- QUADWORD
- QUADWORDS
- RANDOM
- RANDOM_GAUSSIAN
- RANDOM_POISSON
- RANDOM_UNIFORM
- REAL
- RECEIVE
- RECEIVES
- REGION
- REPETITION
- REPETITIONS
- REQUIRE
- RESET
- RESETS
- ROOT
- ROUND
- SECOND
- SECONDS
- SEND
- SENDS
- SIZED
- SLEEP
- SLEEPS
- SQRT
- STANDARD
- STRIDE
- SUCH
- SUM
- SYNCHRONIZATION
- SYNCHRONIZE
- SYNCHRONIZES
- SYNCHRONOUSLY
- TASK

- TASKS
- THAN
- THAT
- THE
- THEIR
- THEM
- THEN
- TIME
- TIMES
- TO
- TORUS_NEIGHBOR
- TORUS_COORDINATE
- TOUCH
- TOUCHES
- TOUCHING
- TREE_PARENT
- TREE_CHILD
- UNALIGNED
- UNIQUE
- UNSUSPECTING
- VALUE
- VARIANCE
- VERIFICATION
- VERSION
- WARMUP
- WHILE
- WHO
- WITH
- WITHOUT
- WORD
- WORDS
- XOR

A.2 Predeclared variables

coNCePTuaL predeclares a set of variables that programs can use but not redeclare. These variables and their descriptions are listed below.

`bit_errors`
: Total number of bit errors observed

`bytes_received`
: Total number of bytes received

`bytes_sent`
: Total number of bytes sent

`elapsed_usecs`
: Elapsed time in microseconds

`msgs_received`
: Total number of messages received

`msgs_sent`
: Total number of messages sent

`num_tasks`
: Number of tasks running the program

`total_bytes`
: Sum of bytes sent and bytes received

`total_msgs`
: Sum of messages sent and messages received

As should be evident from their descriptions, coNCePTuaL's predeclared variables are updated dynamically. Each access can potentially return a different result.

Appendix B Backend Reference

B.1 Method calls

The following method calls must be defined when writing a coNCePTuaL backend from scratch. They are invoked indirectly as part of SPARK's AST traversal. See Section 6.2 [Backend creation], page 94, for more information.

- `n_aggregate_expr`
- `n_aggregate_func`
- `n_all_others`
- `n_an`
- `n_assert`
- `n_awaits_completion`
- `n_backend`
- `n_block_stmt`
- `n_buffer_number`
- `n_byte_count`
- `n_comes_from`
- `n_complex_stmt`
- `n_complex_stmt_list`
- `n_compute_aggregates`
- `n_computes_for`
- `n_data_multiplier`
- `n_data_type`
- `n_even`
- `n_expr`
- `n_for_count`
- `n_for_each`
- `n_for_time`
- `n_func_call`
- `n_header_decl`
- `n_ident`
- `n_ifelse_expr`
- `n_if_stmt`
- `n_in_range`
- `n_integer`
- `n_item_size`
- `n_let`
- `n_let_binding`

- n_let_binding_list
- n_log_expr_list
- n_logs
- n_mcast_stmt
- n_message_alignment
- n_message_spec
- n_no_touching
- n_odd
- n_op_divides
- n_op_power
- n_outputs
- n_processor_stmt
- n_program
- n_random_stride
- n_range_list
- n_real
- n_receive_attrs
- n_receive_stmt
- n_resets
- n_send_attrs
- n_send_stmt
- n_simple_stmt
- n_sleeps_for
- n_stride
- n_string
- n_string_or_expr_list
- n_string_or_log_comment
- n_such_that
- n_sync_stmt
- n_task
- n_task_all
- n_then
- n_time_unit
- n_top_level_complex_stmt
- n_touching
- n_touching_type
- n_touch_stmt
- n_unique
- n_verification
- n_version

B.2 C hooks

To save the backend developer from having to implement coNCePTuaL backends entirely from scratch, coNCePTuaL provides a '`codegen_c_generic.py`' module which defines a base class for code generators that output C code. The base class handles the features that are specific to C but independent of any messaging library. Derived classes need only define those "hook" functions that are needed to implement library-specific functionality.

Hooks are named after the method from which they're called but with an all-uppercase tag appended. The following list shows each hook-calling method in '`codegen_c_generic.py`' and the set of hooks it calls. See Section 6.2.1 [Hook methods], page 95, for more information.

- `code_declare_datatypes`
  - `code_declare_datatypes_EXTRA_EVENTS`
  - `code_declare_datatypes_EXTRA_EVENT_STATE`
  - `code_declare_datatypes_EXTRA_EVS`
  - `code_declare_datatypes_MCAST_STATE`
  - `code_declare_datatypes_POST`
  - `code_declare_datatypes_PRE`
  - `code_declare_datatypes_RECV_STATE`
  - `code_declare_datatypes_SEND_STATE`
  - `code_declare_datatypes_SYNC_STATE`
  - `code_declare_datatypes_WAIT_STATE`
- `code_declare_globals`
  - `code_declare_globals_EXTRA`
- `code_def_alloc_event`
  - `code_def_alloc_event_DECLS`
  - `code_def_alloc_event_POST`
  - `code_def_alloc_event_PRE`
- `code_def_exit_handler`
  - `code_def_exit_handler_BODY`
- `code_def_finalize`
  - `code_def_finalize_DECL`
  - `code_def_finalize_POST`
  - `code_def_finalize_PRE`
- `code_define_functions`
  - `code_define_functions_INIT_COMM_1`
  - `code_define_functions_INIT_COMM_2`
  - `code_define_functions_INIT_COMM_3`
  - `code_define_functions_POST`
  - `code_define_functions_PRE`

- code_define_macros
  - code_define_macros_POST
  - code_define_macros_PRE
- code_define_main
  - code_define_main_DECL
  - code_define_main_POST_EVENTS
  - code_define_main_POST_INIT
  - code_define_main_PRE_EVENTS
  - code_define_main_PRE_INIT
- code_def_init_cmd_line
  - code_def_init_cmd_line_POST_ARGS
  - code_def_init_cmd_line_POST_PARSE
  - code_def_init_cmd_line_PRE_ARGS
  - code_def_init_cmd_line_PRE_PARSE
- code_def_init_decls
  - code_def_init_decls_POST
  - code_def_init_decls_PRE
- code_def_init_init
  - code_def_init_init_POST
  - code_def_init_init_PRE
- code_def_init_misc
  - code_def_init_misc_EXTRA
  - code_def_init_misc_PRE_LOG_OPEN
- code_def_init_msg_mem
  - code_def_init_msg_mem_EACH_TAG
  - code_def_init_msg_mem_POST
  - code_def_init_msg_mem_PRE
- code_def_init_reseed
  - code_def_init_reseed_BCAST
- code_def_init_seed
  - code_def_init_seed_POST
  - code_def_init_seed_PRE
- code_def_procev_arecv
  - code_def_procev_arecv_BODY
- code_def_procev_asend
  - code_def_procev_asend_BODY
- code_def_procev
  - code_def_procev_DECL
- code_def_procev_etime

  - code_def_procev_etime_REDUCE_MAX
- code_def_procev
  - code_def_procev_EVENTS_DECL
  - code_def_procev_EXTRA_EVENTS
- code_def_procev_mcast
  - code_def_procev_mcast_BODY
- code_def_procev_newstmt
  - code_def_procev_newstmt_BODY
- code_def_procev
  - code_def_procev_POST
  - code_def_procev_PRE
  - code_def_procev_PRE_SWITCH
- code_def_procev_recv
  - code_def_procev_recv_BODY
- code_def_procev_repeat
  - code_def_procev_repeat_BODY
- code_def_procev_send
  - code_def_procev_send_BODY
- code_def_procev_sync
  - code_def_procev_sync_BODY
- code_def_procev_wait
  - code_def_procev_wait_BODY_RECVS
  - code_def_procev_wait_BODY_SENDS
- code_def_small_funcs
  - code_def_small_funcs_POST
  - code_def_small_funcs_PRE
- code_output_header_comments
  - code_output_header_comments_EXTRA
- code_specify_include_files
  - code_specify_include_files_POST
  - code_specify_include_files_PRE

B.3 Representing aggregate functions

The LOG_AGGREGATE enumerated type, defined in '`ncptl.h`', accepts the following values:

`NCPTL_FUNC_NO_AGGREGATE`
: Log all data points.

`NCPTL_FUNC_MEAN`
: Log only the arithmetic mean.

`NCPTL_FUNC_HARMONIC_MEAN`
: Log only the harmonic mean.

`NCPTL_FUNC_GEOMETRIC_MEAN`
: Log only the geometric mean.

`NCPTL_FUNC_MEDIAN`
: Log only the median.

`NCPTL_FUNC_STDEV`
: Log only the standard deviation.

`NCPTL_FUNC_VARIANCE`
: Log only the variance.

`NCPTL_FUNC_SUM`
: Log only the sum.

`NCPTL_FUNC_MINIMUM`
: Log only the minimum.

`NCPTL_FUNC_MAXIMUM`
: Log only the maximum.

`NCPTL_FUNC_FINAL`
: Log only the final measurement.

`NCPTL_FUNC_ONLY`
: Log any data point, aborting if they're not all identical.

`NCPTL_FUNC_HISTOGRAM`
: Log a histogram of the data points

License

Summary

This is a BSD license with the additional proviso that modified versions of coNCeP-TuaL must indicate that they are, in fact, modified.

Index

#

-

/

_

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X